D10

```
  2   /*****************************************************
  3   **
  4   **
  5   **      File Name:     RSTinitfin.c
  6   **
  7   **      Copyright (c) 1998,1999 by EMC Corporation.
  8   **
  9   **      Purpose:
 10   **                  This module contains the Restore API functions to
 11   **                  initialize and terminate the restore operation.
 12   **
 13   **      Table of Contents:
 14   **      ** Functions:
 15   **                  API Functions:
 16   **                      EDRMIST_Initialize
 17   **                      EDRMIST_Finish
 18   **
 19   **                  Internal Functions:
 20   **
 21   **
 22   **      Compile-Time Options:
 23   **                  This section must list any compile time definitions
 24   **                  which will affect this header.
 25   **
                                                                          *****/

 28   /* The following provides an RCS id in the binary that can be located
 29      with the what(1) utility.  The intent is to keep this short.
 30   */

 32   #ifndef lint
 33   static char RCS_id [] =    "RCSfiles "
 34                             "$Revision$ "
 35                             "$Date$ "
 36                             ;
 37   #endif

 39   /*
 40    * Feature test switches.
 41    * Standard defines required to turn on OS features go here.
 42    *
 43    * The following is required for code that uses POSIX API's.
 44    * Remove for non-POSIX, non-portable code.
 45    */

 47   #define _POSIX_SOURCE 1

 49   /*
 50    * System headers.
 51    */
 52   #include <pwd.h>

 54   /*
 55    * Epoch headers.
 56    */
 57   #include <eb/eb_port.h>
 58   #include <eb/rb_log.h>
 59
 60   /*
 61    * Local headers
 62    */
 63   #include <RSTinitfin.h>
 64   #include <RSTinternals.h>
```

```
 65   #include <RSTstartup_csm.h>

 67   /*
 68    * Comms headers.
 69    */
 70   #include <restore/csc_EDRMdispatch.h>
 71   #include <restore/csc_EDRMRestoreEng.h>
 72   #include <restore/dispatch_daemon.h>
 73   #include <restore/restore_engine.h>
 74   #include <dmlink/edmlink_api.h>

 76   /*
 77    * #defines, structures, typedefs local to this source file
 78    */

 81   /*
 82    * Global declarations
 83    */

 85   internalHandlePtr handlePtr = NULL;
```

```
 87  *
 88  *  EDMRST_Initialize:
 89  *
 90  *  This function takes care of all the initialization for a recovery
 91  *  session.  This must be called prior to any of the other functions
 92  *  in the Recover API.
 93  *
 94  *  Parameters:
 95  *
 96  *    svrHdl      (I) - The machine name of the server to use.
 97  *
 98  *    hostname
 99  *
100  *    timeout     (O) - A handle to receive a pointer to this user's client
101  *                      handle for the Restore Engine connection.
102  *
104  *                (I) - The maximum number of seconds to wait for the connection
105  *                      to the Restore Engine processes to be completed.
106  *
107  *******************************************************/
108
109  errno_ty
109  EDMRST_Initialize( hostname_ty     hostname,
                        serverhandle    *svrHdl,
                        unsigned long   timeout )
111  {
111      errno_ty  api_status = E_SUCCESS;
112
115      uid_t      human_uid;
116      struct     passwd  *pw;
117      char       *human_uidname ;
118
119      RE_initialize_args    re_init_args;
120      RE_status_result      re_status_result;
121      rpc_if_handle_t       re_if_spec;
122      rpc_binding_handle_t  re_handle;
123      int                   re_index;
124      time_t                end_time;
125
129  #ifdef DEBUG
129  #define  RPC_TIMEOUT       3600
130      struct timeval   rpc_timeout;
131  #endif
132
133  /*********** BEGINNING OF Dispatch Daemon STUFF ***********/
136
137      time( &end_time );  /* compute time to give up waiting */
137      end_time += timeout;
138
138      memset(&if_spec,0,sizeof(rpc_if_handle_t));
139      memset(&re_if_spec,0,sizeof(rpc_if_handle_t));
140
140      if (svrHdl == NULL || hostname == NULL) {
142
143          return ( EP_RB_RECOVER_BAD_ARGS );
144      }
145
146      rec_epi_log_begin( "edmrestore_api" );
147                      /* init logs, ignore errors? */
```

```
149      /* get user name to pass to DD and RE */
149      human_uid = getuid();
150      pw = getpwuid( human_uid );
151      if (pw == NULL) { human_uid = pw->pw_name }
152
153                                    /* Trouble. */
154          rec_epi_log_csm(SUB,CSM_USER_NOT_IN_PASSWD, NULL);
155          return(EP_RB_RECOVER_PERMISSION_DENIED);
156      }
157
158      human_uidname = pw->pw_name;
160
162      handleaddr = (internalhandle *) calloc(1, sizeof(internalHandle));
164      CLIENT_IF_SPEC(if_spec);
165
167      /* Use this macro to setup the interface spec */
169
170      /*
170      **  Arrive at a server binding.   Note that if they didn't give us
171      **  a valid host parameter, this will fail and drop through and
172      **  return NULL in the end.
173      **  This call will get and store a fully resolved binding
174      **  the first time we ever call it to a host.
175      **  csc_get_handle will resolve and store the host binding.  If we
176      **  ever use csc_get_handle to talk to the same host again,
177      **  it will just give back the previously resolved binding.
178      */
179
181      reval = csc_get_handle((unsigned char *) hostname,
182                             if_spec,
183                             SERVER_GROUP,
185                             &handleAttr -> dd_binding_handle,
186                             &status);
188
189      if (status != error_status_ok || (reval == 0)) {
190          /*
190          **  Find out if we got csc_handle and see if status is bad.
191          **  error_status_ok is a macro defined in cscomm.h.
192          */
193
196          /*  If errno not set, use status if it is a valid errno value */
196
198          if ( errno == 0 )
199              errno = ( strerror( status ) ? status : EtIME );
201
203          rec_epi_log_csm( SUB_CSM_RPC_FAIL,
                              "failure finding edmsgpd to start restore
                                                                   engine" );
205
206          return EP_RB_RECOVER_SERVERFAIL;
207      }
208
209      errno = 0 ;
211
211  #ifdef DEBUG
211      /*  setup rpc_timeout during debugging  */
212      rpc_timeout.tv_sec = RPC_TIMEOUT;
213      rpc_timeout.tv_usec = 0;
            cint_control( handleAttr->dd_binding_handle,
                          (char *)&rpc_timeout );
      #endif

            initargs.service = DD_SERVICE_RESTORE;
            initargs.hostname = hostname;
            initargs.username = human_uidname;
```

```c
214   1    initargs.timeout = timeout;

216   1    initres = dd_initialize_1(
217   1            &initargs, handlePtr -> dd_binding_handle );
218   1            /* will have _1 for RPC call */

219   1    if (initres == NULL)
220   1    {
221   1        return EP_RB_RECOVER_RPC_FAIL;
222   1    }

224   1    statargs.service_handle = initres -> service_handle;
225   1    statargs.status = 0;

227   1    statres = dd_getservicestatus_1( &statargs,
                     handlePtr->dd_binding_handle );

229   1    if (statres == NULL)
230   1    {
231   1        return EP_RB_RECOVER_RPC_FAIL;
232   1    }

234   1    while (statres -> status == DD_SERVICE_STARTING )
235   1    {
236   1        time_t now;

238   2        xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
239   2        time( &now );
240   2        if (now >= end_time)
241   2        {
242   3            xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
243   3            "failure waiting for edmdispd to start restore
                     engine" );
244   3            return EP_RB_RECOVER_SERVERFAIL;
245   2        }

247   2        sleep(1);

249   2        statres = dd_getservicestatus_1( &statargs,
250   2                handlePtr -> dd_binding_handle );

252   2        if (statres == NULL)
253   2        {
254   2            rec_api_log_comn( SUB_CSM_RPC_FAIL,
255   2            "failure getting status from edmdispd while starting
                     restore engine" );
256   3            return EP_RB_RECOVER_RPC_FAIL;
258   2        }
260   2    }

261   1    return EP_RB_RECOVER_RPC_FAIL;

262   2    memcpy( handlePtr -> opaque128,
263   2            statres -> handle.handle_val,
264   2            sizeof(handlePtr -> opaque128) );

265   2    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
266   2    return EP_RB_RECOVER_SERVERFAIL;

268   1    }

269   1    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );

270   1    }

272   1    xdr_free( xdr_DD_getservicestatus_result, (char *)statres );
```

```c
274   /* ****************** END OF Dispatch Daemon STUFF ****************** */

276   /* Restore Engine FUNCTIONALITY BEGINS HERE */

278   /*    RE_CLIENT_IFSPEC(re_if_spec); */

280       retval = csc_private_ifspec_init(
281               (unsigned char *) handlePtr -> opaque128,
282               RE_PROGNUM,
283               RE_VERSNUM,
284               &re_if_spec,
285               &status);

287       if (retval == 0)
288       {
289           rec_api_log_comn( SUB_CSM_RPC_FAIL,
290           "failure initializing interface to restore engine" );
291           return EP_RB_RECOVER_SERVERFAIL;
292       }

294       api_status = RE_RECOVER_SERVERFAIL;
295       do {
296           time_t now;
297           time( &now );
298           if (now >= end_time)
299           {
300               rec_api_log_comn( SUB_CSM_RPC_FAIL,
301               "timeout connecting to restore engine" );
302               return EP_RB_RECOVER_SERVERFAIL;
303           }

305           sleep( 1 );  /* give restore engine time to get going */

306           re_handle = csc_connect_to_rpc_service(
307                       (unsigned char *)hostname,
308                       RE_CLIENT_GROUP,
309                       &re_if_spec,
310                       handlePtr ->
311   #ifdef DEBUG
312               /* increase rpc timeout during debugging */
313                       rpc_timeout.tv_sec = RE_RPC_TIMEOUT;
314   #endif
                           re_binding_handle);

316           if ((status == error_status_ok) && (retval == 0))
317           {
318               api_status = E_SUCCESS;
319               re_handle = handlePtr -> re_binding_handle;
320           }

321       } while (api_status != E_SUCCESS);

322       rpc_timeout.tv_sec = RE_RPC_TIMEOUT;
323       rpc_timeout.tv_usec = 0;
324       clnt_control( re.handle, CLSET_TIMEOUT,
325                     (char *)&rpc_timeout );

326       if (api_status == E_SUCCESS)
327       {
328           re_init_args.username = human_uidname;

329           set_rpc_init_result = re_initialize( &re_init_args, &RE_COOKID );
330           if (!re_init_result) {
331               rec_api_log_comn( SUB_CSM_RPC_FAIL,
                  "failure communicating with restore
```

```
332  2                            }
333  1                    else {
334  3                            api_status = re_init_result->status;
335  3                            /* release RPC result struct. */
336  3                            xdr_free( xdr_RE_status_result, (
                                        char *)re_init_result);
337  2                    }
338  1            }
340  1            else
341  1                    rec_api_log_cmt( SUB_CSM_RPC_FAIL,
342  1                            "failure connecting to restore engine" );

344  1            if ( api_status == E_SUCCESS)    /* return rest eng handle on success */
345  1                    *svrHdl = (serverHandle)re_handle;

347  1            return( api_status );
349  1    }   /* End of EDMRST_Initialize() */
```

```
351   /****************************************************************
352   *
353   *  Ping:
354   *
355   *     This function allows a ping to be issued in order to keep the
356   *     engine alive and running so that the engine will not time out.
357   *
358   *  Parameters:
359   *
360   *     srvHdl (I) - A pointer to this user's client handle for the
361   *                  Restore Engine (server) connection.
362   *
363   ****************************************************************/
307 1 errno_ty EDMRST_Ping( serverHandle srvHdl )
365 1 {
366 1     RE_null_args       re_ping_args;
367 1     RE_status_result   *re_ping_result = NULL;
368 1
369 1     if ( NULL == srvHdl
370 2          || srvHdl != handlePtr->re_binding_handle )
371 2     {
372 1        errno_ty      api_status = R_SUCCESS;
373 1        return( EP_RB_RECOVER_BAD_ARGS );
374 1     }
375 1
376 1     set_rpc_obj( re_ping, &re_ping_args RPCobjID );
377 2     re_ping_result = re_ping_( &re_ping_args, srvHdl );
378 2     if ( NULL == re_ping_result )
379 2     {
380 2        api_status = EP_RB_RECOVER_RPC_FAIL;
381 1        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
382 1     }
383   1     else {
384 2        api_status = re_ping_result->status;
385 2        /* release RPC result struct. */
386 1        xdr_free( xdr_RE_status_result, (char *)re_ping_result);
387 1     }
388
```

```
392   /****************************************************************
393   *
394   *  EDMRST_Finish
395   *
396   *  Function Description:
397   *
398   *     This function terminates a restoral session,
399   *     This routine will clean up any local memory used in the session
400   *     disconnect from the Restore Engine,
401   *     EDMRST_Initialize MUST be called before calling any other
402   *  API.
403   *
404   *  Parameters:
405   *
406   *     srvHdl (I) - A pointer to this user's client handle for the
407   *                  Restore Engine (server) connection.
408   *
409   *  Return Codes:
410   *        EP_RB_RECOVER_BAD_ARGS
411   *        EP_RB_RECOVER_RPC_FAIL
412   *        EP_RB_RECOVER_INVALOP
413   *        EP_RB_RECOVER_SERVERFAIL
414   *
415   */
416   errno_ty EDMRST_Finish( serverHandle srvHdl )
417 1 {
418 1     RE_null_args       re_finish_args;
419 1     RE_status_result   *re_finish_result = NULL;
420 1     int                csc_status;
421 1
422 1     if ( NULL == srvHdl
423 1          || srvHdl != handlePtr->re_binding_handle )
424 1     {
425 1        errno_ty      api_status = R_SUCCESS;
426 1        return( EP_RB_RECOVER_BAD_ARGS );
427 1     }
428 1
429 1     set_rpc_obj( re_finish, &re_finish_args RPCobjID );
430 1     re_finish_result = re_finish_( &re_finish_args, srvHdl );
431 1     if ( NULL == re_finish_result )
432 1     {
433 1        api_status = EP_RB_RECOVER_RPC_FAIL;
434 1        rec_api_log_csm( SUB_CSM_RPC_FAIL, NULL);
435 1     }
436   1     else {
437 2        api_status = re_finish_result->status;
438 2        /* release RPC result struct. */
439 2        xdr_free( xdr_RE_status_result, (char *)re_finish_result);
440 1     }
441 1
443 1     rec_api_log_end();    /* write last log and close the log file. */
444 1
445 1     return( api_status );
446   }  /* EDMRST_Finish */
```

```c
%/*
%* Copyright 1997,1998 EMC Corporation
%*/

/*
** Leading % causes rpcgen to pass a line directly through to the output.
** In admin.subroc.h in this case.  This allows the .h to make a little
** more sense and be properly documented.
*/

/*
** dispatch_daemon.x : EDM Dispatch Daemon C/S communication module
*/

/*
** Mission Statement:    This is an RRCGEN file which defines the RPC interface
**                       between the Dispatch Daemon (which resides on the
**                       the EDM server) and the backup client callers of its
**                       functions.  This defines the RPC level calls that a
**                       'caller' can make and the RPC level 'service' will respond to.
**
** Primary Data Acted On: This defines the data that will flow over the wire.
**                        The RPC mechanism will take care of data
**                                      marshalling
**
** Compile-Time Options:  This actually gets run through RRCGEN not compiled.  It
**                        must be run through with the -h flag to create a
**                        header, the -m flag to create the server side
**                        routines, the -l flag to create the client side
**                        routines, and the -c flag to create the common data
**                        marshalling routines.
**
** Basic idea here:       Define the RPC level interfaces to the Dispatch Daemon.
**                        and all data types that will be passed via RPC.
*/

/*******************************************************
Constant Definitions
********************************************************/

/*******************************************************
Data Structure Definitions
********************************************************/

struct DD_rpc_objid
{
    int          type;       /* Object identifier (DD_OTYPE_*) */
};

#define DD_OTYPE_INIT_IN   1     /* Initialize Input Object */
#define DD_OTYPE_INIT_OUT  2     /* Initialize Output Object */

struct DD_client_session_id {
    long         len;        /* Length of structure, version number */
    unsigned long   high;
    unsigned long   low;
};

const DD_SERVICE_RESTORE=1;
/* structures for input and output of re-initialize rpc call. */
struct DD_initialize_args {
    int          service;
};
```

```c
                        string       hostname<>;
                        string       username<>;
                        unsigned int timeout;
                };

const DD_SERVICE_FAILURE_NOEXEC=-4;
const DD_SERVICE_FAILURE_PERMS=-2;
const DD_SERVICE_FAILURE_EXEC=-1;
const DD_SERVICE_STARTING=1;
const DD_SERVICE_RUNNING=2;
const DD_SERVICE_RUNNING=2;
const DD_SERVICE_COMPLETED=4;

struct DD_initialize_result {
    int          status;
    DD_client_session_id   service_handle;
};

/* structures for getstatus function */
struct DD_getservicestatus_args {
    DD_client_session_id   service_handle;
};

struct DD_getservicestatus_result {
    int          status;
    int          handler=128;
};

struct DD_getservicestatus_result {
    int          status;
    opaque       ...;
};

/* work item type */

/*
** These match the rpcconfig.h for the most part.  There are
** some extras for identifying NOS workitems.
*/
const FS_BACKUP_TYPE             = 0;
const FULL_DRAFT_BACKUP_TYPE     = 1;
const SHARED_M_PART_BACKUP_TYPE  = 2;
const OFFLINE_DB_TYPE            = 3;
const ONLINE_NICKKM_TYPE         = 4;
const ONLINE_LISTDB_TYPE         = 5;
const DCONN_SICK_TYPE            = 6;
const DCONN_NET_TYPE             = 7;
const DCONN_WRK_TYPE             = 8;

/* length of various buffers */
const MEDNAME_SIZE    = 6;
const TRLNAME_SIZE    = 16;
const WINAME_SIZE     = 64;
const TAPENAME_SIZE   = 64;
const USERNAME_SIZE   = 64;
const HOSTNAME_SIZE   = 256;
const CLNTNAME_SIZE   = 64;
const SERVNAME_SIZE   = 16;
const DCONN_NAME_SIZE = 16;
const MAX_STRING_SIZE = 256;   /* must be the length of the longest buffer */

/* defines for operation_type */
const EXCPT_TYPE      = 1;
const RESTORE_TYPE    = 2;
const OTHER_TYPE      = 16;

/* work item structure */
struct WIProgress {
    unsigned long    time_started;
    unsigned long    curr_time;
    unsigned long    total_kbytes_sofar;
    unsigned long    total_files;
};
```

```c
struct CC_Notify
{
    char    host_name[HOSTNAME_SIZE];
};

struct EDMStatus
{
    unsigned long   status;
    edm;
    WIProgress      *wiprogress;
};

struct EDMProgress
{
    unsigned long   time_started;
    unsigned long   curr_time;

    unsigned long   total_kbytes;
    unsigned long   total_files;
    unsigned long   total_badfiles;

    unsigned long   curr_time_slice;
    unsigned long   curr_kbytes_sofar;
    unsigned long   curr_files;

    unsigned long   total_files_expected;
    unsigned long   total_kb_expected;

    int     active;
    int     failed;
    int     successful;

    operation_type;
    completed;
    status;

    char    *next;
};

/* SUMMARY structure */
struct EDMProgress {
    unsigned long   total_kbytes_sofar;
    unsigned long   total_files;
    unsigned long   total_badfiles;

    unsigned long   curr_time_slice;
    unsigned long   curr_kbytes_sofar;
    unsigned long   curr_files;

    char    *next;

    operation_type;
    completed;
    status;
};

struct WIProgress
{
    char    wi_name[WINAME_SIZE];
    char    trail_name[TRLNAME_SIZE];
    char    trailee_name[TRLNAME_SIZE];
    char    template_name[TEMPNAME_SIZE];
    char    client_name[CLNTNAME_SIZE];
    char    server_name[SERVER_SIZE];
    char    media_type[MEDNAME_SIZE];
    char    user_id[USERNAME_SIZE];

    char    level;
    char    type;
};
```

```c
{
    int     msgtype;
    int     sourcemodule;
    int     msglen;
    string  msgtext<>;
};

struct SessionInfo
{
    int     DD_client_session_id;
    unsigned int    status;
    unsigned long   jobstarttime;
    long    operation_type;
    long    lastSent;
    long    lastReceived;
    int     outhandle;
    int     errhandle;

    service_handle;
    status;

    SessionInfo     *next;
};

struct SessionBlock
{
    int     sessi;
    struct SessionInfo  totalsessions;
};

program EDM_DISPATCH_DAEMON {
    version EDMDD_FUNCTIONS {
        /* Functions for EDMRST_Initialize */
        DD_initialize_result DD_initialize( DD_initialize_args ) = 1;

        DD_getservicestatus_result DD_getservicestatus(
            DD_getservicestatus_args ) = 2;

        SessionBlock dd_getsessioninfo( dd_getservicestatus_args ) = 3;
    } = 1; /* This is version 1 */
}
/* This is the RPC program number. These are reserved in /pds/docs/RPC_numbers
 * This number cannot be re-used by any other RPC daemon on the machine, as it
 * identifies this daemon uniquely. If it were to be re-used, the last daemon
 * to register would be contacted when RPC's come in for this number.
 */
= 390015;
```

```
1   /*
2   **      Copyright 1996, 1997 EMC Corporation
3   **
4   **      EDMDispatch.c
5   **
6   **
7   ** Mission Statement: This is the main service file for the dispatch
8   **                    daemon.
9   **
10  **                    This file contains the callbacks from the main
11  **                    function
12  **                    which prepares the daemon to go off and service
13  **                    RPC's.
14  **
15  ** Primary Data Acted On:
16  **
17  ** Compile-Time Options:
18  **
19  **      None.
20  **
21  ** Basic idea here: Module for UNIX specific daemon initialization
22  **
23  ** The following provides an RCS id in the binary that can be located
24  ** with the what(1) utility.  The intent is to keep this short.
25  #if !defined(lint)
26  static char   RCS_id [] =  "@(#)$RCSfile: EDMDispatch.c,v $ "
27                             "$Revision: 1.23 $ "
28                             "$Date: 1997/02/05  20:49:15 $" ;
29  #endif
30
31  /* #define _POSIX_SOURCE        unable to compile with this #define set */
32  #define _XOPEN_SOURCE           unable to compile with this #define set */
33
34  #include <esl/c_portable.h>
35  #include <esl/ep_xopen.h>
36  #include <esl/input.h>
37
38  #include <stdarg.h>
39  #include <string.h>
40  #include <syslog.h>
41  #include <pthread.h>
42  #include <thread.h>
43  #include <sys/resource.h>
44
45  #include <logging/logging.h>
46  #include <util/esl_core.h>
47  #include <util/esl_pidfile.h>
48  #include <util/esl_daemon.h>
49  #include <csc/cscomm.h>
50
51  #include <restore/csc_EDMDispatch.h>
52
53  #include <EDMmain.h>
54  #include <EDMDD_ccw.h>
55  #include <EDMDispatchlog.h>
56  #include <EDMDispatchbackground.h>
57  #include <EDMDoc_rstsvc.h>
58
59  /*
60  ** Need to define _XOPEN_SOURCE for signal funtion definitions
61  ** and certain signal structure definitions.
62  */
63  #define _XOPEN_SOURCE
```

```
65  #include <signal.h>
66
67  #undef _XOPEN_SOURCE
68
69  static rpc_if_handle_t if_spec;
70
71  static int          G_debug = FALSE;   /* Variable which will disable forking */
72
73  static char         **commandlineargs; /* pointer to command line args */
74
75  /*******************************************************************
76  **
77  ** Routine:  IsDebugOn
78  **
79  ** Inputs:  None
80  **
81  ** Outputs:  None
82  **
83  ** Return Codes:  TRUE if debug is on.
84  **
85  ** Purpose:       This routine can be used to tell other subsystems
86  **                whether debugging is available.
87  **
88  **
89  ** Intended caller:  Internal only.
90  **
91  **
92  */
93  boolean_ty
94  IsDebugOn()
95  {
96      return G_debug;
97  }
```

```
 99   /* ***************************************************************
100   **
101   **  Routine:  kill_handler
102   **
103   **  Inputs:   int signal  - the signal which was received.
104   **
105   **  Outputs:  Will log messages telling what action is being taken.
106   **
107   **  Return Codes:  exits with the number of the signal received
108   **
109   **
110   **  Purpose:  This routine handles specific signals i.e. SIGINT,
111   **            SIGQUIT, SIGTERM.  Each results in a log entry and an exit.
112   **
113   **
114   **  Intended caller:  internal only.
115   **
116   ** *****************************************************************
117   */
118   static void kill_handler( IN int signal )
119   {
120   int        status;
121   time_t     current_time;
122   char       *ctimebuf;
123   char       *ebuf;

124 1    /* If main exits, it calls this routine with signal 0 */

126 1    /* Unregister the interface */
127 1    (void) csc_unregister_server_interface(&if_spec, &status);

130 1    /* If the unregister fails, report the problem, but continue */
131 1    if ( status != error_status_ok )
132 2    {
132 2       ebuf = (char *) csc_get_error( status );
134 2       (void) EDMDispatch_logent(
               FILE,    LINE,   LOG_ERR,  MESSAGE_NO_LOGIN,  0,
135            "CSC_SERVER_LOGIN failed: <%s> %s",
136 2          status, (ebuf ? ebuf : "Unknown error") );
137 2    }

139 1    /* Get the current time */
140 1    (void) time(&current_time);

142 1    /* Overlay newline with null - buf should always be 26 bytes long */
144 1    ctimebuf = ctime(&current_time);
145 1    ctimebuf[ strlen(ctimebuf) - 1 ] = 0;

147 1    (void) EDMDispatch_logent(
             FILE,    LINE,   LOG_INFO,  MESSAGE_SHUTDOWN,  0,
149          "Shutting down at %s due to signal %d", ctimebuf,
150          signal);

152 1    /* Remove our lock file. */
153 1    (void) EalDestroyPidFile(PIDPATH);

155 1    exit(signal);

157   }  /* End of kill_handler() */
```

```
159   /* ***************************************************************
160   **
161   **  Function Name:
162   **     display_usage
163   **
164   **  Simply displays the usage
165   **
166   **  Call Arguments:
167   **     Program name
168   **
169   **  Error Outputs and Side Effects:
170   **     Prints usage.
171   **
172   **  Special Considerations:
173   **     None.
174   **
175   ** *****************************************************************
176   */
177   static void
178   display_usage (IN char *programe)
179   {
180 1    /* Print out usage */
181 1    fprintf (stderr, "Usage: %s [-d]\n", programe);
182 1    fprintf (stderr, "-d keep the daemon from forking so debugging is easier\n");

184   }  /* end display_usage () */
```

```
187  /*****************************************************************
188  **
189  ** Routine:  daemon_catch_interrupts
190  **
191  ** Inputs:
192  **     None
193  **
194  ** Outputs:
195  **     None
196  **
197  ** Return Codes:
198  **     None
199  **
200  ** Purpose:   Sets up signals for service.  On NT we will have to
201  **            consider what OS constructs to replace signals with.
202  **            In this case we are catching SIGTERM, SIGINT, and
203  **            SIGQUIT and ignoring anything else.
204  **
205  ** Intended caller:  internal only.
206  **
207  *****************************************************************
208  */
209  void daemon_catch_interrupts()
210  {
211      struct sigaction    sactions;       /* signal actions */
213      /*
214       * Set an empty list so we can set signals we want to handle
215       */
216      (void) sigemptyset( &sactions.sa_mask );
218      /*
219       * Add signals that we want to handle
220       */
221      (void) sigaddset( &sactions.sa_mask, SIGTERM );
222      (void) sigaddset( &sactions.sa_mask, SIGINT );
223      (void) sigaddset( &sactions.sa_mask, SIGQUIT );
225      /* Setup the signal handler. */
226      sactions.sa_handler = kill_handler;
228      /*
229       * Assign handler to each signal we are interested in.
230       */
231      (void) sigaction( SIGTERM, &sactions, NULL );
232      (void) sigaction( SIGINT, &sactions, NULL );
233      (void) sigaction( SIGQUIT, &sactions, NULL );
235      /*
236       * Setup mask so we can specify what signals we
237       * will ignore.
238       */
240      /*
241       * We want to ignore everything except those we have set up
242       * above so remove those from the list.
243       */
244      (void) sigfillset( &sactions.sa_mask );
245      (void) sigdelset( &sactions.sa_mask, SIGTERM );
246      (void) sigdelset( &sactions.sa_mask, SIGINT );
248      (void) sigdelset( &sactions.sa_mask, SIGQUIT );
```

```
249      /*
250       * Set the mask.  Since no other threads have been started,
251       * all threads will get this mask.
252       */
253      (void) thr_sigsetmask( SIG_SETMASK, &sactions.sa_mask, NULL );
         }
```

```
256   /**************************************************************
257   **
258   ** Routine: daemon_check_proper_ID
259   **
260   ** Inputs:      None
261   **
262   ** Outputs:     None
263   **
264   ** Return Codes:
265   **              exits with an error when the user is not root
266   **
267   ** Purpose:    Checks user's ID and determines if the user is allowed
268   **             to execute service.
269   **             If there are no constraints then this
270   **             function may be blank.
271   **
272   ** Intended caller:  internal only.
273   **
274   ****************************************************************
275   */
276   void daemon_check_proper_ID()
277   {
278 1     /*
279 1     ** Check for root
280 1     */
281
282 1     if (geteuid() != E_ROOTUID)
283 2     {
284 2         (void) EDMDispatch_logent(
285 2             __FILE__, __LINE__, LOG_ERR, DAEMON_NOTSUPERUSER, 0,
286 2             "Must be run as superuser, uid was %d",
287 3             geteuid());
288 2         exit(1);
289 1     }
      }
```

```
291   /**************************************************************
292   **
293   ** Routine: parse_commandline
294   **
295   ** Inputs:    argc, argv (command line arguments)
296   **
297   ** Outputs:   None
298   **
299   ** Return Codes:
300   **            exits with an error when the user types a bad argument
301   **
302   ** Purpose:   Parses command line arguments and sets flags. If there
303   **            are no flags to be set then this function may be empty.
304   **
305   ** Intended caller: internal only.
306   **
307   **
308   ****************************************************************
309   */
310   void parse_commandline(int argc, char *argv[])
311   {
312 1     int    opt;
313
314 1     commandlineargs = argv;
315
316 1     while ((opt = getopt(argc, argv, "d")) != EOF )    /* Process options */
317 2     {
318 2         switch(opt)
319 3         {
320 3         case 'd':
321 3             g_debug = TRUE;
322 3             break;
323 3         default:
324 3             (void) display_usage( argv[0] );
325 3             exit(1);
326 3         }
327
328 2     }
329 1  }
      }
```

```
332  /***********************************************************
333  **
334  **  Routine: daemon_initialize_logging
335  **
336  **  Inputs:       None
337  **
338  **  Outputs:      None
339  **
340  **  Return Codes:  None
341  **
342  **  Purpose:      Do whatever it takes to initialize logging.  In the near
343  **                future this may involve doing something with catalogs
344  **                or
345  **                calling higher level logging functions which encapsulate
346  **                these things.
347  **
348  **  Intended caller:  internal only.
349  **
350  ***********************************************************/
351  void
353  daemon_initialize_logging ()
354  {
355      /* Pass in argv[0], the program name */
356      (void) esl_log_init(commandlineargs[0]);
358  }
```

```
360  /***********************************************************
361  **
362  **  Routine: daemon_become_daemon
363  **
364  **  Inputs:       None
365  **
366  **  Outputs:      None
367  **
368  **  Return Codes:  None
369  **
370  **  Purpose:      This function is for doing the forking etc. under UNIX.
371  **                It is unknown what will be necessary under NT.
372  **
373  **
374  **  Intended caller:  internal only.
375  **
376  ***********************************************************/
377  void
379  daemon_become_daemon ()
380  {
381      char *ptr;
383      int  ret = 0;

385      /*
386      ** Strip the path from the program name so we can use it
387      ** elsewhere.
388      */
389      ptr = strrchr(commandlineargs[0], '/');
390      if (ptr == NULL)
391          ptr = commandlineargs[0];
392      else
393          ptr++;

395      /* Change directory to a process specific core directory */
396      ret = esl_coredir_setup(ptr);
397      if (ret != 0)
398      {
399          (void) EDMDispatch_logext( __FILE__, __LINE__, LOG_ERR,
400                        MESSAGE_ERR_IN_ESL_COREDIR, 0,
401                        "esl_coredir_setup failed <%s>",
402                        errno);
403          exit(1);
404      }

406      /*
407      ** This is now esl functionality.  This code does everything necessary
408      ** to make this a "real" daemon by detaching from the terminal
409      ** changing the process group, closing stdout, stderr, stdin,
410      */
411      ret = esl_daemon_startup();
412      if (ret != 0)
413      {
414          if (G_debug == FALSE)
415          {
416              fprintf(
417                  stderr, "%s: Failed to initialize as daemon.\n",
418                  commandlineargs[0]);
```

```
419 3                    }
420 2                }
421 1            }
422 1        }
            exit(1);
```

```
424   /******************************************************************
425   **
426   ** Routine: rpc_init
427   **
428   ** Inputs:        None
429   **
430   ** Outputs:       None
431   **
432   ** Return Codes:
433   **      exits with an error code if initialization fails
434   **
435   ** Purpose:   This function is for doing RPC initialization.
436   **            For the most part it involves calling the csc routines.
437   **            This is pretty standard between UNIX and NT.
438   **
439   ** Intended caller:   internal only.
440   **
441   **
442   ******************************************************************
444   */
445 1 void rpc_init()
446 1 {
447 1   error_status_t              status;           /* error status */
449 1   char       *ebuff;                            /* error status (nbase.h) */

451 1 #ifdef _STRUCT_TIMEVAL
452 1   struct timeval      sleep_interval = {5,0};   /* 5 second sleep interval */
453 1 #else
454 1   struct timespec sleep_interval = {5,0};       /* 5 second sleep interval */
455 1 #endif

456 1   /* Setup the interface specification for RPC */
457 1   SERVER_IFSPEC(if_spec);

459 1   /*
460 1    * Login as SERVER_PRINCIPAL.  The context of the process
462 1    * will be set to this principal.
463 1    *
464 1    * This process will keep trying to login to DCE if the
465 1    *                                             registry
466 1    * server is unavailable.  Note that under SUN RPC this is a no-op.
467 1    */
468 1   while (TRUE)
469 1   {
470 2       (void) csc_server_login(SERVER_PRINCIPAL,
471 2                               SERVER_KEYTAB, &status);

472 2       /* If we succeeded, then exit this loop. */
474 2       if ( status == error_status_ok )
475 2       {
476 3           break;
477 3       }
478 2       else /* Print error message if appropriate. */
479 2       {
480 3           ebuff = (char *) csc_csc_error( status );
481 3
```

```
483  3              (void) EDMDispatch_logent(
484  3                          __FILE__, __LINE__, __LOG_ERR,
485  3                          MESSAGE_NO_LOGIN, 0,
486  3                          "CSC_SERVER_LOGIN failed: <%s>",
                                status, (
                                ebuf ? ebuf : "Unknown error"));
487  2          }

489  2          /* If the failure was due to unavailable client,
490  2           * pause and then try again.
491  2           */
492  2          if (status == sec_rgy_server_unavailable)
493  2          {
494  2              /*
495  3               * uses sleep when SUNRPC, otherwise uses
496  3               * pthread call to delay for the specified
497  3               * time
498  3               */
499  3              CSC_SLEEP(sleep_interval);
500  3              continue;
501  2          }

503  2          /* If we got here, we had a unexpected failure. */
504  2          (void) EDMDispatch_logent( __FILE__, __LINE__, __LOG_ERR,
505  2                          MESSAGE_NO_LOGIN, 0,
506  2                          "The service cannot log in as
                                required");

508  2      }
509  1  }

511  2  /*
512  1  ** We need to initialize the authorization module before we do
513  1  ** a listen.
514  1  */
515  1  (void)csc_authorization_init(&status);

517  1  if ( status != error_status_ok )
518  2  {
519  2      ebuf = (char *) csc_get_error( status );

521  2      (void) EDMDispatch_logent( __FILE__, __LINE__, __LOG_ERR,
522  2                      MESSAGE_NOAUTHORIZATION, 0,
523  2                      "CSC_AUTHORIZATION_INIT failed: <%s>",
524  2                      status,
                            ebuf ? ebuf : "Unknown error") );
525  1      exit(1);
526  1  }

529  1  (void) csc_register_server_interface(     &if_spec,
530  1                      SERVER_ANNOTATION,
531  1                      &status);

533  1  if ( status != error_status_ok )
534  2  {
535  2      ebuf = (char *) csc_get_error( status );

537  2      (void) EDMDispatch_logent( __FILE__, __LINE__, __LOG_ERR,
538  2                      MESSAGE_CANNOTREGISTER, 0,
539  2                      "CSC_REGISTER_SERVER_INTERFACE failed:
                                <%d> %s",
540  2                      status, (
```

```
                            ebuf ? ebuf : "Unknown error") );
541  2  }
542  3      exit(1);
543     }
```

```c
545
546   /**********************************************************
547   **
548   ** Routine:   rpc_run
549   **
550   ** Inputs:    None
551   **
552   ** Outputs:   None
553   **
554   ** Return Codes:   None
555   **
556   **
557   ** Purpose:   This function is for running the RPC listen.
558   **            This is pretty standard between UNIX and NT.
559   **
560   ** Intended caller:   Internal only.
561   **
562   */
563
564   void rpc_run()
565   {
566   error_status_t        status;
567
568   char    *ebuff;                          /* error status */
569
570       /* listen for RPC calls forever. */
571       (void) csc_server_listen(
572           rpc_c_listen_max_calls_default, &status );
573
574   ebuff = (char *) csc_get_error( status );
575
576       /* We don't expect to get here. */
577       (void) EDMDispatchLogEvent(
578           _FILE_, _LINE_, LOG_ERR, MESSAGE_SERVERLISTEN, 0,
                 "CSC_SERVER_LISTEN failed: <%d> %s",
                 status,
                 ebuff ? ebuff : "Unknown error" );
         }
```

```c
580   /**********************************************************
581   **
582   ** Routine:   daemon_specific_initialization
583   **
584   ** Inputs:    None
585   **
586   ** Outputs:   None
587   **
588   ** Return Codes:   None
589   **
590   **
591   ** Purpose:   Do whatever makes this daemon special.
592   **                                 In some cases you
593   **            may want to start a thread or open a socket.
594   **                                 Do that here.
595   **
596   ** Intended caller:   Internal only.
597   **
598   */
599
600   void
601   daemon_specific_initialization()
602   {
603   error_status_t    status;               /* error status */
604
605   int           ret;
606   pthread_t     mant_id;
607   pthread_t     cleanid;
608   time_t        current_time;
609   char          *ctimebuf;
610   struct rlimit rlp;
611
612       /* Create a file and lock it so we don't start multiple
613       ** daemons.  Exit if there is another copy of us running.
614       ** The CreatePIDFile call already logs errors so just exit.
615       */
616       if (&sCreatePidFile(PIDPATH))
617       {
618           exit(1);
619       }
620
621       /* Find out what time it is */
622       (void) time(&current_time);
623
624       ctimebuf = ctime(&current_time);
625
626       /* Overlay newline with null - but should always be 26 bytes
627       ** long */
628       ctimebuf[ strlen(ctimebuf) - 1 ] = 0;
629
630       /* Log startup message */
631       (void) EDMDispatchLogEvent(
632           _FILE_, _LINE_, LOG_INFO, MESSAGE_STARTUP, 0,
                 "Restore service %s starting up at %s",
                 commandline,argv[0],
                 ctimebuf );
633
634       /* set the open files limit very large */
635       rlp.rlim_max = FD_SETSIZE;
636       rlp.rlim_cur = FD_SETSIZE;
637       setrlimit(RLIMIT_NOFILE, &rlp);
```

```
638 1        getrlimit(RLIMIT_NOFILE, &rlp);
640 1
        (void) EDMDispatch_logent(
641 1            __FILE__, __LINE__, LOG_INFO, MESSAGE_STARTUP, 0,
                 "Service allows %d open files",
                 rlp.rlim_max );

        /* Initialize service launcher */
641 1   ret = EDMDSSvcInit(1);
644 1

646 1   if (ret != 0)
647 2   {
648 2       (void) EDMDispatch_logent(
                __FILE__, __LINE__, LOG_INFO, 0, 0,
                "Service Launcher failed returning - %d",
                ret);
649 2
650 2       exit(1);
651 1   }

        /*
653 1    * Start the other threads in the daemon. The main thread
654 1    * becomes the RPC thread. BANDrxMessage is the entry point
655 1    * for the data collection thread. BANDtxCleanup is the
656 1    * entry point for the data expiration thread.
657 1    */
658 1   /*pthread_create(&wantid, NULL, DispDaemon_crr, NULL);*/
659 1   pthread_create(&wantid, NULL, DispDaemon_ccw, NULL);
660 1   pthread_create(&clientid, NULL, DispatchBackground, NULL);
661 1   rpc_init();
662 1   rpc_run();
663 1
664 1  }
```

```
666     /***********************************************************
667     **
668     ** Routine: daemon_cleanup
669     **
670     ** Inputs:      None
671     **
672     ** Outputs:     None
673     **
674     ** Return Codes:
675     **              None
676     **
677     ** Purpose:     Call function which will clean up daemon properly.
678     **
679     ** Intended caller: internal only.
680     **
681     **********************************************************/
682     */
684     void
685     daemon_cleanup()
686     {
687         kill_handler( 0 );
688     }
```

```
 1   /*
 2   ** Copyright 1996,1997 EMC Corporation
 3   */
 4
 5   /*
 6   ** EDMDispatchService.c
 7   **
 8   ** Mission Statement:     RPC entry points.
 9   **
10   ** Primary Data Acted On:
11   **
12   ** Compile-Time Options:
13   **
14   ** Basic idea here:
15   */
16
17   #if !defined(lint)
18   static char   RCS_id [] = "@(#)$RCSfile: EDMDispatchService.c,v $ *
19                             "$Revision: 1.0 $ *
20                             "$Date: 1997/02/06 20:49:15 $" ;
21   #endif
22
23
24   #include <esl/c_portable.h>
25   #include <esl/inout.h>
26
27   #include <logging/logging.h>
28   #include <csc/cscomm.h>
29
30   #include <restore/csc_EDMDispatch.h>
31   #include <restore/dispatch_daemon.h>
32
33   #include <EDMDispatching.h>
34   #include <EDMDispatchSession.h>
35
36   /*
37   ** These are all the rpc entry points for the dispatch daemon.
38   ** The dispatch daemon is multi-threaded and it is the main thread
39   ** which handles all incoming RPC.  ONC RPC is single threaded
40   ** which blocks other RPC calls.  This provides us some
41   ** safety in the way we handle our data, and limits our exposure
42   ** to unexpected multithreading problems.
43   */
44   static void FreeSessionInfo(SessionInfo *);

46

47   /*
48   ** ***************************************************************
49   ** Routine:       dd_initialize_1
50   ** Inputs:        DD_initialize_args * - args for the restore session initialize
                                             call
51   **
52   ** Outputs:       None
53   ** Return Codes:
54   **    DD_initialize_result * - result of init function call
55   **
56   ** Purpose:  Function to create & restore session.
57   **
58   ** Intended caller: Internal Only.
59   ** ***************************************************************
60   */
61

63   DD_initialize_result *
```

```
64   dd_initialize_1_svc(  IN DD_initialize_args *arg, IN struct svc_req *req )
65 1 {
66 1   static DD_initialize_result argzz;

68 1   InitializeSession(arg, req, &argzz);

70 1   return &argzz;
71 1 }
```

```c
 73  /*****************************************************************
 74  **
 75  ** Routine:    dd_getservicestatus_1
 76  **
 77  ** Inputs:     dd_getservicestatus_args * - args for the
 78                                              getservicestatus call
 80  ** Outputs:    None
 81  **
 82  ** Return Codes:
 83  **    DD_getservicestatus_result * - result of status function
 84                                       call
 85  **
 86  ** Purpose:    Function to poll for status on a session.
 87  **
 88  ** Intended caller: Internal Only.
 89  **
 90  *****************************************************************/
 90  DD_getservicestatus_result *
 91  dd_getservicestatus_1_svc(
 91      IN DD_getservicestatus_args *arg, IN struct svc_req *req )
 92  {
 92      static DD_getservicestatus_result argzzz;
 93
 95      GetDispatchStatus(arg, &argzzz);
 97      return &argzzz;
 98  }
```

```c
100  /*****************************************************************
101  **
102  ** Routine:    dd_getsessioninfo_1
103  **
104  ** Inputs:     dd_getsessioninfo_args * - args for the getsessioninfo
105                                            call
106  ** Outputs:    None
107  **
108  ** Return Codes:
109  **    DD_getsessioninfo_args * - result of session info call
110  **
111  **
112  ** Purpose:    Function to get information on all sessions.
113  **
114  ** Intended caller: Internal Only.
115  **
116  *****************************************************************/
117  SessionBlock *
118  dd_getsessioninfo_1_svc(
118      IN DD_getservicestatus_args *arg, IN struct svc_req *req )
119  {
119      static SessionBlock argzzz;
120      static boolean_ty first = TRUE;
121
123      if (first)
124      {
125          memset(&argzzz, 0, sizeof(argzzz));
126          first = FALSE;
127      }
128      else
129      {
130          FreeSessionInfo(&argzzz.sess);
131          argzzz.sess = NULL;
132      }
134      GetDispatchInfo(arg, &argzzz);
136      return &argzzz;
137  }
```

```
119    /************************************************
140    **
141    **    Routine:    FreeSessionInfo
142    **
143    **    Inputs:     SessionInfo * - arg to free
144    **
145    **    Outputs:    None
146    **
147    **    Return Codes:
148    **                None
149    **
150    **    Purpose:    Function to free all SessionInfo structures in a list.
151    **
152    **    Intended caller:  Internal Only.
153    **    ********************************************/
155 1  static void FreeSessionInfo(SessionInfo *sess)
156 1  {
157 1      if (sess == NULL)
158 1          return;
160 1      if (sess -> next != NULL)
161 1          FreeSessionInfo(sess -> next);
163 1      free(sess);
164 1  }
```

```
  1   /*
  2   **     Copyright 1996, 1999 EMC Corporation
  3   **
  4   **
  5   **     EDMDispatchSession.cc
  6   **
  7   **     Mission Statement: This is where all session management occurs.
  8   **
  9   **
 10   **     Primary Data Acted On:
 11   **
 12   **
 13   **     Compile-Time Options:
 14   **
 15   **             USR_SUNRPC - Compile source with sunrpc
 16   **                          support. If not set, assume DCE support.
 17   **
 18   **     Basic idea here: Module for session management
 19   */
 20   /*
 21   **     This following provides an RCS id in the binary that can be located
 22   **     with the what(1) utility.  The intent is to keep this short.
 23   */
 24   #if defined(lint)
 25   static char    RCS_id [] = "@(#)$RCSfile: EDMDispatchSession.cc,v $ *
 26                              $Revision: 1.3 $ *
 27                              $Date: 1997/02/05 20:49:15 $ ";
 28   #endif
 29
 30   /*  #define _POSIX_SOURCE                 unable to compile with this define set */
 31   #define _SVID_SOURCE                      unable to compile with this define set */
 32
 33   #include <esi/c_portable.h>
 34   #include <esi/ep_wopen.h>
 35   #include <esi/input.h>
 36
 37   #include <pthread.h>
 38   #include <memory.h>
 39   #include <sys/time.h>
 40   #include <sys/types.h>
 41   #include <syslog.h>
 42
 43   // Rogue Wave includes
 44   #include <rw/collect.h>
 45   #include <rw/rwfile.h>
 46   #include <rw/vstream.h>
 47   #include <rw/bintree.h>
 48
 49   #include <csc/cscomm.h>
 50   #include <restore/dispatch_daemon.h>
 51   #include <restore/dispatch_protocol_client.h>
 52   #include <EDMSession.h>
 53   #include <EDMDReturnMessageapi.h>
 54   #include <EDMDHandleHprm1.h>
 55   #include <EDMDispatchSession.h>
 56   #include <EDMDispatchConfig.h>
 57   #include <EDMDDot_retsvc.h>
 58   #include <EDMDispatchLog.h>
 59
 60   static RWBinaryTree             G_sessionTree;
 61
 62   static pthread_mutex_t          G_sessionMutex = PTHREAD_MUTEX_INITIALIZER;
 63   extern BlinkHandle*_ty          BlinkHandle;
 64
```

```
 65   static int maxDisconnectTime = SECONDS_PER_HOUR;  // one hour
 66
 67   /***********************************************************************
 68   **
 69   **     Routine:        LockSessionMutex
 70   **
 71   **     Inputs:         None
 72   **
 73   **     Outputs:        None
 74   **
 75   **     Return Codes:
 76   **                     None
 77   **
 78   **     Purpose:        Lock the session mutex.
 79   **
 80   **
 81   ************************************************************************/
 82
 83   static void
 84   LockSessionMutex()
 85   {
 86       static boolean_ty first = TRUE;
 87
 88       if (first == TRUE)
 89       {
 90           first = FALSE;
 91           pthread_mutex_init(&G_sessionMutex, NULL);
 92       }
 93
 94       pthread_mutex_lock(&G_sessionMutex);
 95   }
 96
```

```
 98
 99
100   /***********************************************************
101    **
102    **   Routine:      UnlockSessionMutex
103    **
104    **   Inputs:       None
105    **
106    **   Outputs:      None
107    **
108    **   Return Codes:
109    **                 None
110    **
111    **   Purpose:      Unlock the mutex for the session tree object
112    **
113    *********************************************************
114    */
115   static void
116   UnlockSessionMutex()
117   {
118       pthread_mutex_unlock(&G_sessionTreeMtx);
119   }
```

```
120   /***********************************************************
121    **
122    **   Routine:      DD_initialize
123    **
124    **   Inputs:       DD_initialize_args *arg - args sent via RPC for starting
125    **                                           session
126    **                 struct svc_req *req - the request block from RPC
127    **
128    **   Outputs:      DD_initialize_result *res - the result structure which
129    **                                  operation succeeded or failed.  tells whether
130    **
131    **   Return Codes:
132    **                 None
133    **
134    **   Purpose:      Initialize a session for the GUI.
135    **
136    *********************************************************
137    */
138   void
139   InitializeSession(IN DD_initialize_args *arg, IN struct svc_req *req,
140                     OUT DD_initialize_result *res)
141   {
142       EDMSession    *session;
143       EDMSession    *ret;
144       pthread_t     id;
145       time_t        t;
146
147       if (arg == NULL || req == NULL || res == NULL)
148       {
149           return;
150       }
151
152       t = time(NULL);
153
154       session = new EDMSession();
155       if (session == NULL)
156       {
157           res -> status = DD_SERVICE_FAILURE_NONEXEC;
158           return;
159       }
160
161       session -> initSession();
162
163       session -> setStartTime(t);
164
165       session -> setOperationType(arg -> service);
166
167       session -> setStatus(DD_SERVICE_STARTING);
168
169   #if 0
170       if (arg -> username != NULL && arg -> hostname != NULL)
171       {
172           switch(arg -> service)
173           {
174               // code is commented out because we do not
175               // want to read the config for permission information
176               // at this time, it is a waste of cycles
177               case DD_SERVICE_RESTORE:
178                   allowed = boolean_by allowed;
179
180   #if 0
```

```
181  3        DispatchCheckRestorePermission(
                       arg->hostname,
                       arg -> username);

183  3
184  4
185  4
186  4              if (!allowed)
187  4              {
188  4                  res -> status =
                             DD_SERVICE_FAILURE_PERMS;
                          delete session;
                          return;
190  3              }
191  3
192  3          break;
193  3
193  3 #endif
193  3          default: // Add some error message for unknown service
194  2          break;
195  1              };
196  2
197  2        else
198  2        {
199  2              res -> status = DD_SERVICE_FAILURE_NONEXEC;
199  2              delete session;
200  2              return;
201  1        }
201  1
203  1      LockSessionMutex();
205  1      ret = (EDMSession *) G_sessionTree.insert({
                                 RWCollectable *) session);
207  1
207  1      UnlockSessionMutex();
209  1      if (ret == NULL)
210  2      {
211  2              res ->      status = DD_SERVICE_FAILURE_NONEXEC;
212  2              delete session;
213  2              return;
214  1      }
214  1
216  1      session -> getSessionID(&res -> service_handle);
218  1
218  1      // Call Steve's thread
219  1      pthread_create(&id, NULL, &DDRSrvc_init, (void *) session);
221  1
221  1      session -> setThreadID(id);
223  1
223  1      return;
224  1    }
```

```
226        /****************************************************
227        **
228        **  Routine:     SendPingMessagesToSession
229        **
230        **  Inputs:      None
231        **
232        **  Outputs:     None
233        **
234        **  Return Codes: None
235        **
236        **  Purpose:     Queue up all the ping messages to the sessions.
237        **               respond they should be considered dead. If they don't
238        **
239        **
240        **
241        */
241
242        void
243        SendPingMessagesToSession()
244        {
245  1        EDMSession *sess;
246  1
248  1        LockSessionMutex();
250  1        RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                          G_sessionTree);
252  1
252  1        while ( sessionIterator != NULL &&
253  1               (sess = (EDMSession*) (*sessionIterator)()) != NULL)
253  1        {
254  2              DD_client_session_id sid;
255  2              rpc_binding_handle_t *cscb = NULL;
256  2              int                   status;
257  2              int                   ret;
258  2
260  2              if (sess -> getStatus() != DD_SERVICE_RUNNING)
261  2                  continue;
261  2
263  2              sess -> getSessionID(&sid);
265  2              ret = GetCSCBHandle(&sid, &cscb, &status);
267  2              if (ret != 0 || cscb == NULL || *cscb == NULL)
268  2                  continue;
270  2
271  2              PushResponseMessage(dp_ping_request, sid, cscb, &status);
271  2
273  2              // through with iterator
274  2              if (sessionIterator != NULL)
275  2              {
276  2                  delete sessionIterator;
277  2              }
277  2        }
279  1        UnlockSessionMutex();
280  1    }
```

```
282  /*********************************************************
283  **                                                     *******
284  **                                                     *******
285  **  Routine:   UpdateSessionLastReceived               *******
286  **                                                     *******
287  **  Inputs:   DD_client_session_id *sessID - session that sent us
288  **                                                    something
289  **                                                     *******
290  **  Outputs:   None                                    *******
291  **                                                     *******
292  **  Return Codes:                                      *******
293  **       0 on success and non-zero otherwise           *******
294  **                                                     *******
295  **  Purpose:   Update the specified session with the lastest received
296  **                                                    message
297  **                                                     *******
298  **             time.                                   *******
299  **                                                     *******
300  **                                                     *******
301  */
302
303  int
304  UpdateSessionLastReceived(DD_client_session_id *sessID)
305  {
306  1      time_t       last = time(NULL);
307  1      EDMSession  *session;
308  1      EDMSession  *ret;
309  1
310  1      session = new EDMSession();
311  1
312  1      if (session == NULL)
313  2      {
314  2          EDMDispatch_logent(
315  2              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
316  2              "failure to create a session block");
317  2          return -1;
318  1      }
319  1
320  1      session -> setSessionID(sessID);
321  1
322  1      LockSessionMutex();
323  1
324  1      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);
325  1
326  1      UnlockSessionMutex();
327  1
328  1      delete session;
329  1
330  1      if (ret == NULL)
331  2      {
332  2          EDMDispatch_logent(
333  2              __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
334  2              "failure to update session %ld:%ld received time",
335  2              sessID -> high, sessID -> low);
336  2          return -1;
337  1      }
338  1
339  1      ret -> setLastReceived(last);
340  1
341  1      return 0;
342  }
```

```
338  /*********************************************************
339  **                                                     *******
340  **                                                     *******
341  **  Routine:   UpdateSessionLastSent                   *******
342  **                                                     *******
343  **  Inputs:   DD_client_session_id *sessID - session that sent us
344  **                                                    something
345  **                                                     *******
346  **  Outputs:   None                                    *******
347  **                                                     *******
348  **  Return Codes:                                      *******
349  **       0 on success and non-zero otherwise           *******
350  **                                                     *******
351  **  Purpose:   Update the specified session with the lastest sent
352  **                                                    message
353  **                                                     *******
354  **             time.                                   *******
355  **                                                     *******
356  **                                                     *******
357  */
358
359  int
360  UpdateSessionLastSent(DD_client_session_id *sessID)
361  {
362  1      time_t       last = time(NULL);
363  1      EDMSession  *session;
364  1      EDMSession  *ret;
365  1
366  1      session = new EDMSession();
367  1
368  1      if (session == NULL)
369  2      {
370  2          EDMDispatch_logent(
371  2              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
372  2              "failure to create a session block");
373  2          return -1;
374  1      }
375  1
376  1      session -> setSessionID(sessID);
377  1
378  1      LockSessionMutex();
379  1
380  1      ret = (EDMSession *) G_sessionTree.find((RWCollectable *) session);
381  1
382  1      UnlockSessionMutex();
383  1
384  1      delete session;
385  1
386  1      if (ret == NULL)
387  2      {
388  2          EDMDispatch_logent(
389  2              __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
390  2              "failure to update session %ld:%ld sent time",
391  2              sessID -> high, sessID -> low);
392  2          return -1;
393  1      }
394  1
395  1      ret -> setLastSent(last);
396  1
397  1      return 0;
398  }
```

```c
/*****************************************************************
 *
 * Routine:    CheckDispatchSessions
 *
 * Inputs:     None
 *
 * Outputs:    None
 *
 * Return Codes:  None
 *
 * Purpose:    Look for dead sessions and kill them off
 *
 *****************************************************************/
void
CheckDispatchSessions()
{
    EDMSession    *sess;
    int           status = 0;
    int           ret = 0;
    time_t        curTime;
    time_L        curTime;
    RWBinaryTree  reaperTree;

    curTime = time(NULL);

    LockSessionMutex();

    RWBinaryTreeIterator *sessionIterator = new RWBinaryTreeIterator(
                                                    Q_sessionTree);

    while ( (sess = (EDMSession*)(*sessionIterator)()) != NULL ) {

        if ( (sess->getLastReceived() != 0 ||
              curTime - sess->getLastReceived() <= curTime - maxDisconnectTime &&
              (sess->getStatus()
              (sess->getStartTime() &&
              curTime - sess->getStartTime() <= curTime - maxDisconnectTime &&
              (sess->getStatus()
                  == DD_SERVICE_FAILURE_NONEXEC ||  sess -> getStatus()
                  == DD_SERVICE_FAILURE_FAILURE_EXEC ||
                     DD_SERVICE_FAILURE_FAILURE_DEMS) )

        {
            // Insert it into the reaper tree
            (void) reaperTree.insert(sess);
        }
    }

    UnlockSessionMutex();

    // If the reaper tree has something in it then use those entries
    // things from the query tree.
    if (reaperTree.entries() > 0)
    {
        sessionIterator = new RWBinaryTreeIterator(reaperTree);
```

```c
    while ( sessionIterator != NULL &&
            (sess = (EDMSession*)(*sessionIterator)()) != NULL ) {
        sess->_client_session_id
        sess->getSessionID(&sessID);

        if (ret != 0)
        {
            ret = removeSession(&sessID);

            if (ret != 0)
            {
                EDMDispatch.logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                    "failure to remove session %ld:%ld",
                    sessID.high, sessID.low);
            }
            continue;
        }
        else
        {
            EDMDispatch.logent( __FILE__, __LINE__, LOG_INFO, 0, 0,
                "removing session %ld:%ld",
                sessID.high, sessID.low);
            Haven't recieved anything since %ld. Current %ld",
            sess->getLastReceived(),
            curTime - maxDisconnectTime)

            ret = deleteHandleSet(&sessID, &ELinkHandle, &status);

            if (ret != 0)
            {
                EDMDispatch.logent( __FILE__, __LINE__, LOG_ERR, 0, 0,
                    "failure to delete handles for session %ld:%ld",
                    sessID.high, sessID.low);
            }
        }
    }

    // through with iterator
    if (sessionIterator != NULL)
    {
        delete sessionIterator;
    }

    reaperTree.clear();
}
```

```
495    /*********************************************************
496    **
497    **   Routine:    DrainSessionDescriptors
498    **
499    **   Inputs:     None
500    **
501    **   Outputs:    None
502    **
503    **   Return Codes:
504    **       None
505    **
506    **   Purpose: Drain whatever data is on stdout and stderr for sessions.
507    **
508    **
509    */
510    void
511    DrainSessionDescriptors()
512    {
512        int     hout = 0, herr = 0, status = 0;
513        int     selret = 0;
514        int     i = 0;
515        char    buff[1024];
516        struct  timeval timetowait = {
517                    1, 0
518                };
519        fd_set  stdoutSet;
520        fd_set  stderrSet;
521        fd_set  stderrSet;
522
524        getStdoutSet(&stdoutSet, &hout, &status);
526        if ( (selret = select(
                      hout + 1, &stdoutSet, NULL, NULL, &timetowait)) >= 0)
527        {
528            for (i = 0; i < hout+i; i++)
529            {
530                if (FD_ISSET(i, &stdoutSet))
531                {
532                    while (read(i, buff, 1024) > 0);
533                }
534            }
535        }
536
537        getStderrSet(&stderrSet, &herr, &status);
538
539        if ( (selret = select(
                      herr + 1, &stderrSet, NULL, NULL, &timetowait)) >= 0)
540        {
541            for (i = 0; i < herr+i; i++)
542            {
543                if (FD_ISSET(i, &stderrSet))
544                {
545                    while (read(i, buff, 1024) > 0);
546                }
547            }
548        }
549    }
```

```
551    /*********************************************************
552    **
553    **   Routine:    GetSessionStatus
554    **
555    **   Inputs:     DD_client_session_id *ssid, int *s_status, int *status
556    **                   session ID to check the status of
557    **
558    **   Outputs:    int  *status   - status of the function call
559    **               int  *s_status - session status
560    **
561    **   Return Codes:
562    **       0 if successful and non-zero otherwise
563    **
564    **   Purpose: Get status on the session.
565    **
566    **
567    */
568    int
569    GetSessionStatus(
570        DD_client_session_id *ssid, int *s_status, int *status)
571    {
572        EDMSession  *sess;
573        EDMSession  *ref;
574
575        if (status == NULL)
576        {
577            return -1;
578        }
579
580        if (ssid == NULL || s_status == NULL)
581        {
582            *status = SESSION_BAD_ARGS;
583            return -1;
584        }
585
586        sess = new EDMSession();
587
588        if (sess == NULL)
589        {
590            *status = SESSION_NO_MEMORY;
591            EDMDispatch_logout(
                     __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
593                     "Failure to create a session block");
594            return -1;
595        }
596
597        LockSessionMutex();
598
599        sess -> setSessionId(ssid);
600
601        ret = (EDMSession *) G_sessionTree.find((RWCollectable *) sess);
602
603        UnlockSessionMutex();
604
605        delete sess;
606
607        if (ret == NULL)
608        {
609            EDMDispatch_logout(
                     __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                     "Failure to lookup session %ld:%ld");
```

```
610 2        *status = SESSION_LOOKUP_FAILED;
                     ssid -> high, ssid -> low);
612 2        return 1;
613 1    }

615 1    *s_status = ret -> getStatus();

617 1    return 0;
618 }
```

```
620    /***************************************************************
621    **
622    **  Routine:    GetDispatchStatus
623    **
624    **  Inputs:     DD_getservicestatus_args *arg - session ID to check the
625                                                    status of
626    **  Outputs:    DD_getservicestatus_result *res - the result structure
                                                        which tells
                        whether operation succeeded or failed.
627    **  Return Codes:
628    **      None
629    **
630    **
631    **
632    **  Purpose:    Get status on the starting session.
633    **
634    **
635    */
636
637    void
638    GetDispatchStatus(IN DD_getservicestatus_args *arg,
639                      OUT DD_getservicestatus_result *res)
640    {
641 1    static char buff[CONNECT_HANDLE_SIZE];
642 2    EDMSession *ret;
642 2    EDMSession *sess;
643 1
645 2    sess = new EDMSession();

647 1    if (sess == NULL)
648 1    { // Give an error
649 2        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                "failure to create a session block");
650 2
651 1        return;
652 1    }

654 2    sess -> setSessionID(&arg -> service_handle);

655 1    LockSessionMutex();

658 2    ret = (EDMSession *) g_sessionTree.find((RWCollectable *) sess);

660 1    UnlockSessionMutex();

662 1    delete sess;

664 2    if (ret == NULL)
665 2    {
666 2        EDMDispatch_logent(
                __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                "failure to lookup session %ld:%ld",
                arg -> service_handle.high,
                arg -> service_handle.low);
667
668
670 2        res -> status = DD_SERVICE_FAILURE_NONEXEC;
671 1        return;
672 1    }

674 1    res -> status = ret -> getStatus();

676 1    memset(buff, 0, sizeof(buff));
```

```
678  2      if (res -> status == DD_SERVICE_RUNNING)
679  2      {
680  2          res -> handle.handle_val = (char *) ret -> getConnectionHandle(
                                                                               );
681  2          res -> handle.handle_len = CONNECT_HANDLE_SIZE;
682  2      }
     else
684  2      {
685  2          res -> handle.handle_val = (char *) buff;
686  2          res -> handle.handle_len = CONNECT_HANDLE_SIZE;
687  2      }
688  1  }
```

```
690     /**********************************************************
691     **
692     ** Routine:     GetDispatchInfo
693     **
694     ** Inputs:      DD_getservicestatus_args *arg - session ID to check the
695     **                                              status of
696     ** Outputs:     SessionBlock *res - the information regarding the
697     **                                  specified session
698     **
699     ** Return Codes:
700     **              None
701     **
702     ** Purpose:     Get status on all the sessions.
703     **
704     ***********************************************************/

706     void
707     GetDispatchInfo(IN DD_getservicestatus_args *arg,
708                     OUT SessionBlock *res)
709  1  {
710  1      EDMSession      *sess;
711  1      EDMSession      *ret;
712  1      SessionInfo     *sinfo, *slast;
713  1      static char buff[CONNECT_HANDLE_SIZE];

715  1      LockSessionMutex();

717  1      if (arg -> service_handle.high != 0 && arg -> service_handle.low != 0)
718  2      {
719  2          // Looking for a single session. Do a find.
720  2          sess = new EDMSession();

722  2          if (sess == NULL)
723  2          {
724  3              // Give an error
725  3              EDMDispatch_logent(
726  3                  __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
727  3                  "Failure to create a session block");
728  3              UnlockSessionMutex();
729  2              return;
730  2          }

731  2          sess = setSessionID(arg -> service_handle);

733  2          ret = (EDMSession *) G_sessiontree.find(sess);

735  2          delete sess;

737  2          if (ret == NULL)
738  2          {
739  3              EDMDispatch_logent(
740  3                  __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
741  3                  "Failure to lookup session %d:%d",
742  3                  arg -> service_handle.high,
743  3                  arg -> service_handle.low);
744  3              UnlockSessionMutex();
                    return;
746  2          }

             res -> totalsessions = 1;
```

```
748  2      res -> sess = (SessionInfo *) calloc(1, sizeof(SessionInfo));

750  2      if (res -> sess == NULL)
751  2      {
752  3          EDMDispatch_logout(
753  3              __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                       "Failure to allocate session info
                        block");

755  1          UnlockSessionMutex();
756  1          return;
757         }

758  2      sinfo = res -> sess;

760  2      ret -> getSessionID(&sinfo -> service_handle);
761  2      sinfo -> status = ret -> getStatus();
762  2      sinfo -> jobstarttime = ret -> getStartTime();
763  2      sinfo -> operation_type = ret -> getOperationType();
764  2      sinfo -> lastSent = ret -> getLastSent();
765  2      sinfo -> lastReceived = ret -> getLastReceived();

767  3          UnlockSessionMutex();
768  3          return;
769         }
        else
        {
771  2      res -> totalsessions = 0;

773  2      sinfo = (SessionInfo *) calloc(1, sizeof(SessionInfo));
774  2      if (res -> sess == NULL)
775  2      {
776  3          EDMDispatch_logout(
                    __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                    "Failure to allocate session info
                     block");

778  3          UnlockSessionMutex();
779  3          return;
            }

781  2      sinfo = res -> sess;

783  2      RWBinaryTreeIterator *sessionIterator = new
                RWBinaryTreeIterator(G_sessionTree);

785  2      boolean_ty addnext = FALSE;

787  2      while ( sessionIterator != NULL && (ret = (EDMSession*) (
                *sessionIterator)()) != NULL )
            {
                int          status;

                if (addnext)
                {
791  2              sinfo -> next = (SessionInfo *) calloc(1, sizeof(
                        SessionInfo));
793  4              if (sinfo -> next == NULL)
794  4              {
                        break;
                    }

                    sinfo = sinfo -> next;
                }

800  6          ret -> getSessionID(&sinfo -> service_handle);
801  3          sinfo -> status = ret -> getStatus();
803  3          sinfo -> jobstarttime = ret -> getStartTime();
```

```
806  3          sinfo -> operation_type = ret -> getOperationType();
807  3          sinfo -> lastSent = ret -> getLastSent();
808  3          sinfo -> lastReceived = ret -> getLastReceived();

810  3          getHandleSet(
811  3              &sinfo -> service_handle, &sinfo -> outhandle,
                        &status);

813  3          res -> totalsessions++;

815  3          sinfo -> next = NULL;
816  3          addnext = TRUE;
817  2      }

819  2      // through with iterator
820  2      if (sessionIterator != NULL)
821  2          delete sessionIterator;
822  2      }
823  2  }

825  1      UnlockSessionMutex();

827  1  }
828  1  }
```

```
810

831    /*************************************************************
832    **
833    ** Routine:   removeSession
834    **
835    ** Inputs:
836    **
837    ** Outputs:
838    **    *ret;
839    **
840    ** Return Codes:
841    **    None
842    **
843    ** Purpose: Remove the active session object between the GUI and the
       **          service.
844    **
       **************************************************************/
846    int
847    removeSession(IN DO_client_session_id *sess_id,
848    **                                    OUT int *status)
849    {
850        EDMSession *sess;
851        EDMSession *ret;
852
854        if (status == NULL)
855        {
856            return -1;
857        }
858
859        if (sess_id == NULL)
860        {
861            *status = SESSION_BAD_ARGS;
862            return -1;
863        }
864
865        *status = 0;
866        if (G_sessionTree.isEmpty())
867        {
868            EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, SESSION_LIST_EMPTY, 0,
                   "No sessions in list.
                   Can't remove session <%id %id>",
                   sess_id -> high, sess_id -> low);
               return -1;
           }

872        sess = new EDMSession();

874        if (sess == NULL)
876        {
877            EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, SESSION_NO_MEMORY, 0,
                   "Failure to create a session block");
878            return -1;
879        }

880        sess -> setSessionID(sess_id);

882
884        LockSessionMutex();

886
888        ret = (EDMSession *) G_sessionTree.remove(sess);
```

```
890        UnlockSessionMutex();

892        if (ret == NULL)
893        {
894            EDMDispatch_logent(
                   __FILE__, __LINE__, LOG_ERR, SESSION_LOOKUP_FAILED, 0,
                   "Failure to remove session %id,%id",
                   sess_id -> high, sess_id -> low);

896            delete sess;
897            *status = SESSION_LOOKUP_FAILED;
898            return -1;
899        }

900
902        delete ret;
903        delete sess;

905        return 0;
906    }
```

D11

```
1    /********************************************************************
2    **
3    **
4    **    File Name:    RSLstart.c
5    **
6    **    Copyright (c) 1998, 1999 by EMC Corporation.
7    **
8    **    Purpose:
9    **
10   **      The intent of the contents of this file is to implement the
11   **      functions the control execution of the restore for the
12   **                                                    Restore Service
13   **      Library.
14   **
15   **      These functions are provided to allow:
16   **        - creation of submit objects,
17   **              which define the set of objects to be
18   **              restored and the scripts to be run before and after
19   **                                                    restoration,
20   **        - starting the restore of a submit object.
21   **
22   **    RSTSL_Start
23   **
24   **    the following functions comprise restore management:
25   **
26   **
27   **                                                    *******/

30   /*
31   ** Feature test switches.
32   ** Standard defines required to turn on OS features go here.
33   ** the following is required for code that uses POSIX API's.
34   ** Remove for non-POSIX, non-portable code.
35   **
36   ** Compile-Time Options:
37   **      This section must list any compile time definitions
38   **      which will affect this header.
     */

     #define _POSIX_SOURCE 1

     /*
     ** System headers.
     */
43   #include <sys/wait.h>

     /*
     ** Epoch headers.
     */
47   #include <eb/eb_port.h>
48   #include <eb/rtl_log.h>
49   #include <eb/utili_normalize.h>
50   #include <util1/ebutil.h>
51   #include <ebreport/ehi.h>

     /*
     ** Local headers
     */
57   #include <RSLinterns.h>
58   #include <restore/StandardSupp.h>
59   #include <restore/dispatch.h>
```

```
52   #include <RSLinterns.h>
53   #include <restore/StandardSupp.h>
54   #include <restore/dispatchd_daemon.h>
55   #include <restore/RDMREProgress.h>

64   extern int RunExecutable(const boolean_ty ResetuId,
                              const int RunUid,
                              const char *starting_cwd,
                              const char *executable_name,
                              char **executable_argv,
                              char **executable_env,
                              int *run_exit_status,
                              boolean_ty *run_cancelled,
                              boolean_ty (*QuitTest)(void));

80   extern int RunWorkItemRestore(int, boolean_ty (*CancelRestoreTest)());

84   static eerrno_ty
     ExecuteWorkItemRestore(int SubmitObjectID,
                            boolean_ty (*QuitTest)(void));

88   static eerrno_ty
     RunPrepareRestore(int SubmitObjectID,
                       int (*QuitTest)(void),
                       int *rumphase_status,
                       int *cleanupExit);

96   static eerrno_ty
     RunCleanupRestore(int SubmitObjectID,
                       boolean_ty (*QuitTest)(void),
                       int rumphase,
                       int *cleanupExit);

100  /*
101  ** #defines, structures, typedefs local to this source file
102  */
103  #define STR_SIZE(str)  (str) ? str : ""
104  #define REMOVE_NEWLINE(str) \
105  {\
106      int rem_nl_index;\
107      for(rem_nl_index = 0; str[rem_nl_index] != '\0'; rem_nl_index++)\
108          if(str[rem_nl_index] == '\n')\
109              str[rem_nl_index] = '\0';\
110  }\

111  /*
112  ** Start
113  **
114  ** This function begins execution of the restoral of the objects in a
115  ** submit object.  Its progress and requests for operator input are
116  ** returned via RSTSL_GetRestoreFeedback.
117  **
118  ** Parameters:
119  **
120  **   SubmitObjectID  ( I ) - ID of the submit object which describes the restore
121  **   QuitTest        ( I ) - function to call to check for quit signal
122  **
123  ** **************************************************************************/
124  eerrno_ty RSTSL_Start( int SubmitObjectID, boolean_ty (*QuitTest)(void))
125  {
```

```
127 1    int ret_pre;
128 1    int ret_exec;
129 1    int ret_post;
130 1    int ret_all_ok = 0;
131 1    int PrepareExit = 0;
132 1    int QuitExit = 0;
133 1    boolean_t QuitFlag = FALSE;

135 1    char aactime[32];

137 2    memset(aactime, 0, 32);

139 1    (void)ctime(&rcp->rc_cmd_starttime);

141 1    (void)ctime_r(&rcp->rc_cmd_starttime, aactime, 32);

143 1    rcp->rc_last_wri_time = rcp->rc_cmd_starttime;

145 1    REMOVE_NEWLINE(aactime);

147 1    rbe_log_stats(0, "Restore Started at %s.", aactime);

149 1    rbe_log_stats(0, "Restore Started application type %s.",
150 1        (rcp->rc_backup_app == 0)
151 1        ? "Network"
152 1        : (/* struct plugindata */
153 1           rcp->currentPptr-> isDelta)
154 1           name );

155 1    rbe_log_stats(0, "Restore Started of "\
156 1        "rcp level object: %s, template %s.",
157 1        STR_SURE(rcp->rc_top_level_object_name),
158 1        STR_SURE(rcp->rc_template_name),
159 1        (rcp->savesec_thread) ? "Alternate" : "Primary");

161 1    rbe_log_stats(0, "Restore Started by user %s, Uid %d, Gid %d.",
162 1        STR_SURE(rcp->rc_human_uidname),
163 1        rcp->rc_human_uid, rcp->rc_human_gids[0]);
164 1

166 1    rbe_log_stats(0, "Restore Started with client destination %s.",
167 1        STR_SURE(rcp->rc_client_hostname));

169 1    /* if not a network restore,
170 1        check if plugin has its own start function */

171 1    if ( rcp->rc_backup_app != 0
172 1         && NULL != rcp->currentPptr-> pifuncArray)
173 2    {
174 2        ret_exec = rcp->currentPptr-> pifuncArray[
                 pifuncIndexStartRestore ](
175 3            rcp.SubmitObjectID, QuitTest );

178 1        if( QuitTest() )    /* check for abort before return */
179 2        {
180 1            rbe_log_stats( EP_RB_RECOVER_ABORT,
181 1                "The restore was quit by the user during
182 3                execution.");
                     setGlobalStatus( EMMRE_STATE_USER_QUIT );
                     /* set RE's internal status */
```

```
183 3            return EP_RB_RECOVER_ABORT;
184 3        }

186 2        if (E.SUCCESS != ret_exec)
187 2            setGlobalStatus( EMMRE_STATE_FAILED );
                     /* set RE's internal status */
188

189 2        else
                     setGlobalStatus( EMMRE_STATE_SUCCESSFUL );
                     /* set RE's internal status */

191 2        return ret_exec ;
192

193 3    }

195 2    ret_pre = RunPrepareRestore(SubmitObjectID,
                 QuitTest,
                 &PrepareExit);

198 1    if(EP_RB_RECOVER_ABORT == ret_pre)
199 3    {
200 2        rbe_log_stats(EP_RB_RECOVER_ABORT,
201 2            "The restore was quit by the user during
202 2            preparation.");

203 2        setGlobalStatus( EMMRE_STATE_USER_QUIT );
                 /* set RE's internal status */

205 2        return EP_RB_RECOVER_ABORT;
206 2    }
207 1    else if(EP_RB_RECOVER_PREFAILED == ret_pre)
208 2    {
209 2        rbe_log_stats(EP_RB_RECOVER_PREFAILED,
                 "The restore failed during preparation. Exit %d",
                 PrepareExit);

212 2        setGlobalStatus( EMMRE_STATE_PREFAILED );
                 /* set RE's internal status */

215 1        return (EP_RB_RECOVER_PREFAILED);

217 1    }
218 1    if(PrepareExit != 0)
                 QuitFlag = QuitTest();
220 1    }

223 1    if (E.SUCCESS == ret_exec)      /* check if any MIs failed */
224 1    {
225 2        int local_stat;
226 2        EMMStats stats;
227 2        memset( &stats, 0, sizeof(EMMStats) );

229 3        if (0 != getRestoreStatus( 0, &stats, &local_stat ))
230 3        {
231 3            rbe_log_stats(
232 2                local_stat, "internal error: Failed in getRestoreStatus.");
                     ret_exec = ret_all_ok = EP_RB_RECOVER_EXECFAILED;
234 3            }
235 3            else
236 2            {
                     if (stats.edm.failed)     /* Is a failure for cleanup purposes */
                         ret_all_ok = EP_RB_RECOVER_ALLFAIL;
                     if ( stats.edm.successful > stats.edm.failed )
                         ret_all_ok = EP_RB_RECOVER_PREFAIL;
```

```
240  2          }
241  2          else
242  2              ret_all_ok = EF_RB_RECOVER_MANFAIL;
243  2      }

247  1      ret_post = RunCleanupRestore(SubmitobjectID,
248  1                                   QuitTest,
                                         ret_all_ok,
                                         &CleanupExit);

250  2      if(QuitFlag)
251  2      {
252  2          rbe_log_stats(EF_RB_RECOVER_ABORT,
253  2              "The restore was quit by the user before execution.
                                                                     ");
254  2          setGlobalStatus( EDMRE_STATE_USER_QUIT );   /* set RF's internal status */
255  2          return EF_RB_RECOVER_ABORT;
256  1      }

258  1      if (E_SUCCESS != ret_exec)        /* return execute status if it failed
                                                                                 */
259  2          setGlobalStatus( EDMRE_STATE_FAILED );   /* set RF's internal status */
260  2          return ret_exec;
261  2      }

262  1      if(EF_RB_RECOVER_ABORT == ret_post)
264  1      {
265  2          rbe_log_stats(EF_RB_RECOVER_ABORT,
266  2              "The restore was quit by the user during cleanup.");
267  2          setGlobalStatus( EDMRE_STATE_USER_QUIT );
268  2          setGlobalStatus( EDMRE_STATE_USER_QUIT );   /* set RF's internal status */
269  2          return EF_RB_RECOVER_ABORT;
270  1      }

271  1      if( (CleanupExit != 0 ) || (E_SUCCESS != ret_post) )
272  1      {
273  2          rbe_log_stats(EF_RB_RECOVER_POSTFAILED,
274  2              "The restore failed during cleanup. Exit %d",
275  2                  CleanupExit);
276  2          setGlobalStatus( EDMRE_STATE_FAILED );   /* set RF's internal status */
278  2          return (EF_RB_RECOVER_POSTFAILED);        /* set RF's internal status */
279  2      }

281  1      setGlobalStatus( EDMRE_STATE_SUCCESSFUL );       /* set RF's internal status */

283  1      }

285  1      return( E_SUCCESS );

287  1      }          /* RSTSL_Start */
```

```
            static errno_ty
            ExecuteWorkItemRestore(int SubmitobjectID,
                                   boolean_ty (*QuitTest)(void))
            {
                int ret_RunItem;
                sm_push();

297  1
298  1          rcp->error_message[0] = 0;
299  1
301  1          if(0 != (ret_RunItem = RunWorkItemRestore(
                                       SubmitobjectID, QuitTest)))
303  1          {
304  2              rbe_log_stats(0,"Internal error: Failed in RunWorkItemRestore");
305  2          }
306  2          }

308  1          sm_pop();

310  1          if (QuitTest() == TRUE)
311  1              return EF_RB_RECOVER_ABORT;

313  1          if (ret_RunItem != 0)
314  1              return EF_RB_RECOVER_EXECUTEFAILED;

316  1          return E_SUCCESS;
317  1      }
```

```
320   #define EXECUTABLE_MAX 1024
321   static errno_ty
322   RunPrepareRestore(int SubmitObjectID,
323                     boolean_ty (*QuitTest)(void),
324                     int *PrepareExit)
325   {
326 1   char **prephaseargs = NULL;
327 1   char **prephaseenv = NULL;
328 1   int GetStatus = 0;
329 1   char preExecutable[EXECUTABLE_MAX];
330 1   boolean_ty restore_cancelled = FALSE;
331
332 1   *PrepareExit = 0;
333
334 1   /*
335 1    * GetSOPrePhase allocates prephaseargs & prephaseenv.
336 1    * This will need to be free'ed later.
337 1    */
338 1
339 1   if(0 != GetSOPrePhase(SubmitObjectID,
340 2                preExecutable,
341 2                EXECUTABLE_MAX,
342 2                &prephaseargs,
343 2                &prephaseenv,
344 2                &GetStatus))
345 2   {
346 2
347 2       rbe_log_state(0,"Internal error: Failed in GetSOPrePhase");
348 2       return (EP_RB_RECOVER_FATALERR);
349 2   }
350 2
351 1   if(0 != strcmp(preExecutable, ""))
352 2   {
353 2       setGlobalStatus( EDMRE_STATE_PREPHASE );
354 2       if(-1 == RunExecutable(FALSE,
355 2                   preExecutable,
356 2                   0,
357 2                   NULL,
358 2                   prephaseargs,
359 2                   prephaseenv,
360 2                   PrepareExit,
361 2                   &restore_cancelled,
362 2                   QuitTest))
363 2       {
364 3           rbe_log_state(
365 4               0,"Internal error: Failed in RunExecutable for prepare.");
366 3           return (EP_RB_RECOVER_FATALERR);
367 2       }
368 2       if(TRUE == restore_cancelled)
369 2           return (EP_RB_RECOVER_ABORT);
370 1   }
371
372 1   return( E_SUCCESS );
373   }
```

```
375   boolean_ty alwaysFalse() ( return FALSE; )
```

```c
377    static errno_ty
379    RunCleanupRestore(int SubmitObjectID,
380 1      boolean_ty (*QuitTest) (void),
381 1      int runphase_status,
382 1      int CleanupExit)
382 1    {
384 1      char **postphaseargs = NULL;
385 1      char *postphaseenv = NULL;
386 1      int CleanupExit = 0;
387 1      char postExecutable[EXECUTABLE_MAX];
388 1      boolean_ty restore_cancelled = FALSE;
389 1      boolean_ty ignore_quit=FALSE;

391 1      *CleanupExit = 0;

393 1      /*
394 1       * GetSOPostPhase allocates postphaseargs & postphaseenv.
395 1       * This will need to be free'ed later.
396 1       */

398 1      PurgeTrailQueue();
399 1      if(0 != GetSOPostPhase(SubmitObjectID,
400 1                 postExecutable,
401 1                 EXECUTABLE_MAX,
402 1                 &postphaseargs,
403 1                 &postphaseenv,
404 1                 &GetSOStatus))
406 6        rbe_log_status(0,"internal error: Failed in GetSOPostPhase?");
407 6        return (EP_RB_POSTPHASE_FATALERR);
408 1      }

410 1    #define RESTORE_BREAK     "RESTORE_BREAK=TRUE"
411 1    #define RESTORE_NOBREAK   "RESTORE_BREAK="
413 1    #define RESTORE_BREAK_ERROR "RESTORE_BREAK=E"

414 1      if(0 != strcmp(postExecutable, ""))
415 1      {
416 1        char *abort=NULL;
417 1        if(QuitTest())
418 1        {
419 1          /*
420 1           * If a quit has been specified, we need to tweak the
421 1           * environment variable if set
422 1           */
423 2          abort=RESTORE_BREAK;
424 2        }
425 2        else if(0!=runphase_status)
426 2        {
427 2          abort=RESTORE_BREAK_ERROR;
428 2        }

430 2        /*
431 2         * ignore_quit is set to 1 when we have already processed a BREAK
432 2         * (RUNCANCEL on gui) and are using the environment variable to clean
433 2         * RESTORE_BREAK_E to signal the post restore script to clean
434 2         * up from this break.  When that happens, an ignore_quit value
435 2         * of 1 will resuse actual quit signals to be ignored, by the
436 2         * cleanup script, since we already know (via the environment
437 2         * variable) that we are in "cleanup mode" and no further signal
438 2         * interception is necessary.
439 2         */

441 2        ignore_quit=FALSE;
442 3        if(NULL!=abort && NULL!=postphaseenv)
443 3        {
```

```c
442 3        {
443 4          int isub=0;
444 4          char *cptr;
445 4          while(cptr=postphaseenv[isub])
446 5          {
447 5            if(strncmp(postphaseenv[isub],RESTORE_BREAK,strlen(
                      RESTORE_BREAK))==0)
448 5            {
449 6              postphaseenv[isub]=rsl_strdup(abort);
450 6              ignore_quit=TRUE;
451 6              if(NULL==postphaseenv[isub])
452 6              {
453 6                rbe_log_status(
454 6                     0,"internal error: "Allocate failed in RSIstart.c")
455 6                return EP_RB_RECOVER_POSTFAILED;
456 6              }
457 5            }
458 4            isub++;
459 4          }
460 3        }

461 2        setGlobalStatus( EDMEE_STATE_POSTPHASE );   /* set RE's internal status */

462 2        if(-1 == RunExecutable(FALSE,
463 5               NULL,
464 5               postExecutable,
465 5               postphaseargs,
466 5               postphaseenv,
467 5               restore_cancelled,
468 5               CleanupExit,
469 5               ignore_quit=alwaysFalse(QuitTest))
470 5        {
471 6          rbe_log_status(
472 6               0,"internal error: Failed in RunExecutable for cleanup.");
473 6          return (EP_RB_RECOVER_FATALERR);
474 5        }
475 5        if(TRUE == restore_cancelled)
476 5          return (EP_RB_RECOVER_ABORT);
477 5      }

479 1      return( E_SUCCESS );
480 1    }
```

```
482   static eerrno_ty
483   RunExecutionOverrideRestore(int SubmitObjectID,
484                               boolean_ty (*QuitTest)(void))
485   {
487 1    return( E_SUCCESS );
488 }   }
```

```
489   #undef EXECUTABLE_MAX
```

```
1    /*
2    **
3    **   File Name:   RSLwsvr.c
4    **
5    **   Copyright (c) 1998,1999 by BMC Corporation.
6    **
7    **   --------------------------------------------------
8    **   Purpose:
9    **
10   **   The intent of the contents of this file is to implement the
11   **   functions that control execution of work item restores for
12   **   RunWorkItemRestores()
13   **
14   **   Service Library.
15   **
16   **   These functions are provided to allow:
17   **
18   **   The following functions comprise restored management:
19   **
20   **   Compile-Time Options
21   **        this section must list any compile time definitions
22   **        which will affect this header.
23   **
24   **                                                           */

27   #define _POSIX_SOURCE 1

30   /*
31    * System headers.
32    */
34   #include <sys/time.h>
35   #include <sys/types.h>
36   #include <sys/wait.h>
37   #include <values.h>

39   /*
40    * Epoch headers.
41    */
42   #include <eb/eb_port.h>
43   #include <eb/rb_log.h>
44   #include <ebutil/ebutil.h>
45   #include <restore/RSprogMsg.h>

47   /*
48    * Local headers.
49    */
51   #include <RSLagouts.h>
52   #include <RSLsemf.h>
53   #include <EDMRBCledapi.h>
54   #include <EDMRBodleodMsgrpi.h>
55   #include <RSLinterms.h>
56   #include <RSLibshmain.h>
57   #include <restore/EDMRBShmtmtpi.h>
58   #include <restore/EDMRBdrainapi.h>

60   #define STR_SURE(str)   (str) ? str : ""

63   /*
```

```
64   **   RunWorkItemRestores
65   **
66   **   Set the number of drives being used for the life of the restore.
67   **   SeqLclFlag = FALSE;
68   **   TrailRestoresLeft = ( # of trail restores )
69   **   TrailRestoresRunning = 0;
70   **
71   **   while drive available.
72   **      RunTrail -- Set drive concurrency for trail restores.
73   **      TrailRestoresRunning++;
74   **      while OKToRunWIForTrail
75   **         StartWIRestore
76   **
77   **   while(1)
78   **      if(Quitter)
79   **         Send WICancel
80   **         SeqLclFlag = TRUE
81   **      Select(from_pipe, timeout (5 seconds))
82   **      for each WI that completes
83   **         interpreate the return.
84   **         Drain in progress.
85   **         Send final Progress for work item.
86   **         if(Finished)
87   **            if(OKToReschedule && SetQuitFlag == FALSE)
88   **               TrailQueue
89   **            if(TrailRestoresAreMoreWorkItems && SetQuitFlag == FALSE)
90   **               while OKToRunWIForTrail
91   **                  StartWIRestore
92   **            else -- RunMoreTrailRestore
93   **               RunTrailRestore(prevTrailQueue)
94   **               TrailRestoresLeft--;
95   **               while ( drive available &&
96   **                       TrailRestoresLeft-- &&
97   **                       drive available &&
98   **                       SeqLclFlag == FALSE &&
99   **                  RunTrail -- Set drive concurrency for trail restores.
100  **                  TrailRestoresRunning++;
101  **                  while (
102  **                     StartWIRestore
103  **                     OKToRunWIForTrail && SetQuitFlag == FALSE)
104  **                  end while
105  **               end
106  **            end else
107  **      End for each WI completes
108  **      if ((SeqLclFlag == TRUE) && (TrailRestoresRunning == 0)) ||
109  **          ((SeqLclFlag == 0) && (TrailRestoresRunning == 0)) ||
110  **          TrailRestoresLeft == 0)
111  **         return;
112  **      exit loop.
113  **   end while(1)
114  **
115  **   )
116  */

118  /* Stube */
119  static int
120  InterpretWorkItemRestoresResults(wi_restore_results *results);

122  static int
123  test_fd(int fd);

125  static void
126  DebugLogFds(char *error_msg,
127              fd_set *fds);
```

```c
static int
DetermineGlobalDriveUse();

static int
test_fd_hup(int fd);

static int
FindTrailIDForWItem(int handle,
                    int *TrailID,
                    int *status);

static int
SendRunningWorkItemsQuit();

static int
Select(int nfds,
       fd_set readfds,
       fd_set writefds,
       fd_set exceptfds,
       struct timeval *timeout);

/* End Stubs */

static int
HandleWorkItemRestoreResults(int FromFD,
                    int TrailID,
                    WI_restore_results *results);

static int
RunWorkItemRestoresForTrail(const int TrailID,
                    const int CountDrivesAvailable,
                    boolean_ty (*CancelRestoresetOff)(),
                    boolean_ty *QuitFlag,
                    int *CountDrivesInUse);

/*
 * Runs a set of work item restores.
 *
 * Args:
 *   SubmitObject
 *   CancelRestoresetTest().
 *
 * Returns: int 0 for success.
 */

int
RunWorkItemRestores(int SubmitObject, boolean_ty (*CancelRestoresetTest)())
{
    boolean_ty QuitFlag  = FALSE;   /* Has the user requested a quit. */
    boolean_ty GenQuit   = FALSE;   /* Have we initiated the quit. */

    int TrailRestoresRunning = 0;   /* The number of trail restores running. */

    int TrailRestoresTotal;         /* The number of trail restores total. */

    int TrailRestoresLeft;          /* The number of trail restores left. */

    int CountDrivesAvailable = 0;   /* The count of drives available. */
    int CountDrivesInUse = 0;       /* The count of drives in use. */

    int HighestActiveTrail;         /* The trail queues are ordered from 1 to n. */

    int temp_status;
```

```c
    /* This is the highest trail running
     */

    if(debugmode)
    {
        (void)the_user_error(0,
                    "DEBUG: Running RunWorkItemRestores.");
    }

    /*
     * Generate the trail queues.
     * Buckets the work items into trail queues. The trail queues are sorted
     * in the order which the work items should run.
     */

    if(0 != GenerateTrailQueues(SubmitObject,
                    &TrailRestoresTotal,
                    &temp_status))
    {
        (void)the_user_error(0,
                    "Internal error: Cannot generate trail
                    queues, cannot continue.");
        return -1;
    }

    if(debugmode)
    {
        (void)the_user_error(0,
                    "DEBUG: RunWorkItemRestores for %d trails.",
                    TrailRestoresTotal);
    }

    CountDrivesAvailable = DetermineGlobalDriveUse(/*SubmitObject*/);

    TrailRestoresLeft = TrailRestoresTotal;

    /*
     * This is the start up loop to get the initial work item
     * restores started.
     */

    QuitFlag = CancelRestoresetTest();

    while((CountDrivesInUse < CountDrivesAvailable) &&
          (HighestActiveTrail < TrailRestoresTotal) &&
          (FALSE == QuitFlag))
    {
        int submitobjID = 0;
        int submitelementID = 0;

        /*
         * Activate the Trail Queue.
         * This allows the trail queue to be used to
         * determine the work item restores to run.
         */
        if(0 != ActivateTrailQueue(HighestActiveTrail,
                    1,
                    &temp_status))
        {
            (void)the_user_error(0,
                    "Internal error: Cannot activate trail
                    queues(1) for trailid %d, cannot continue.",
                    HighestActiveTrail);
            return -1;
        }

        HighestActiveTrail++;
```

```c
251  2      /*
252  2       * i.e. the count of running work item restores
253  2       * this sets the number of drives and media access concurrency for
                  trail restores
                  for this trail.
                  Today this is one.
                  */
254  2
255  2
256  2
257  3      if(0 != SetQDrivesAcquired(HighestActiveTrail, 1, &temp_status))
258  3      {
259  3          (void)the_user_error(0,
                    "Internal error: Cannot set drive acquired
                    1] for trailid %d, cannot continue.", HighestActiveTrail);
260  3          return -1;
261  3      }
262  3
263  2      if (0 > (temp_status = RunWorkItemRestoresForTrail(
                                     HighestActiveTrail,
                                     CountDrivesAvailable,
                                     CancelRestoreTest,
                                     &QuitFlag,
                                     &CountDrivesInUse)))
264  2      {
265  2
266  2          /* RunWorkItemRestoresForTrail does its own error logging. */
267  2          return -1;
           }
269  3
270  3      if(temp_status == 0)
271  3      {
               (void)the_log_stats(0,
                   "Trail %d restores had no work item to run!
                   1).", HighestActiveTrail);
               /* more work may be need to recover from this error condition. */
273  3      }
274  3
275  3      if(temp_status > 0)
276  3      {
               TrailsRestoresRunning++;
277  3      }
           }/* End while() initial startup loop */
278  1
279  1
280  1
281  1      while(1)
282  1      {
283  1          int HighestFd = 0;
285  1          fd_set WorkItemFromFds;
               int restStatus;
               struct timeval timeout = {5, 0};
288  2          if(QuitFlag && (!SentQuit))
289  2          {
290  2              (void)the_log_stats(0,
                       "Restore was quit by user. Quitting restore,
                       this could take a while.");
292  2          }
293  2
294  3          {
                   SendRunningWorkItemsQuit();
                   SentQuit = TRUE;
295  3          }
297  3
298  3          if(0 != getFromSet(&WorkItemFromFds, &HighestFd, &retStatus))
299  3          {
                   (void)the_user_error(0,
                     "Internal error: Cannot get auxproc result
                     fds, cannot continue.");
```

```c
306  2              return -1;
307  5          }
308  5  #if 0
309  4          if(0 > (retStatus = Select(HighestFd + 1,
                                          &WorkItemFromFds,
                                          NULL, NULL,
                                          &timeout)))
310  2          {
311  2              DebugLogFds("The file descriptors to wait on are ",
                              &WorkItemFromFds);
312  5          }
313  5
315  5  #endif
316  2          if(0 > (retStatus = Select(HighestFd + 1,
                                          &WorkItemFromFds,
                                          NULL, NULL,
                                          &timeout)))
317  2          {
318  2              /* error */
319  2              (void)the_user_error(RBRECOVER_MKERR(errno),
                        "Internal error: Cannot get auxproc result
                        fds, cannot continue.");
320  2
321  2              return -1;
324  3          }
325  2          else if (0 == retStatus)
326  2          {   /* timed out */
327  3              QuitFlag = CancelRestoreTest();
331  3          }
333  2          else
334  3          {   /* Available fds */
336  5              int ReadyFds = retStatus;
337  3              int FoundFds = 0;
338  3              int index;
340  5
341  4              DebugLogFds("The file descriptors ready to read are ",
                              &WorkItemFromFds);
342  4              {
                   /* If there are available fds then we may want to
                    * schedule the next work item from restore. We should
                    * check if the user initiated a quit.
                    */
343  4
346  5                  QuitFlag = CancelRestoreTest();
348  5
349  5                  for(index = 0;
                       (index < (HighestFd + 1)) && (FoundFds < ReadyFds);
                       index++)
350  5                  {
351  5                      int StartWorkItemForTrail = 0;
353  3                      if(FD_ISSET(index, &WorkItemFromFds))
354  4                      {
357  4                          int TrailID;
358  5                          int TrailAcquired;
359  4                          wi_restore_results results;
360  5                          FoundFds++;
362  5
363  5                          memset(&results, 0, sizeof(wi_restore_results));
365  5
367  5                          if(0 != HandleWorkItemRestoreResults(index,
                                                                    &TrailID,
                                                                    &results))
370  6                          {
                               }
```

```
371  6
372  6        /* HandleWorkItemRestoreResults will do its own logging/
373  5
374  5            return -1;
375  5        }
376  5        /*
377  5         * This is where we may want to retry the work item
               * Based on if it passes or fails
               */
379  5        CountDrivesInUse--;

382  5        if (0 > (StartWorkItemRestoresForTrail(TrailID,
                      CountDrivesAvailable,
                      CancelRestoresReset,
                      &QuitFlag,
                      &CountDrivesInUse)))
384  5
385  5
386  5
387  5
388  5
389  5
390  5
391  5        {
                  (void)rbe_user_error(0,
                    "Internal error: Cannot
                    return -1;
393  5        }

               /* RunWorkItemRestoresForTrail does its own logging. */

395  5        else if (StartWorkItemForTrail == 0)
396  5        {
               /* 0 work items started above,
                  lets check to see if this is the last work item for
397  5            this trail
398  5         */
399  5

401  6        int wtCount;
402  6
403  6        if (0 != GetRunningWIT(TrailID, &wtCount, &temp_status))
404  6        {
406  6            (void)rbe_user_error(0,
                   "Internal error: Cannot
407  6            return -1;
408  6        }
409  6        if(debugmode)
410  6        {
411  6            (void)rbe_user_error(0,
                   "DEBUG: RunWorkItemRestores no
                   more work items left for trailid %d,
                   but %d wtCount workitems still running",
                   TrailID, wtCount);
412  6        }
413  6        /*
414  6         * Testing for No work items left running or started
415  6         * For this trail.
416  6         */
418  6        if((0 == wtCount) && (0 == StartWorkItemForTrail))
419  6        {
421  7            TrailRestoresRunning--;
422  7            TrailRestoresLeft--;
423  7
424  7            if(0 != DeactivateTrailQueue(TrailID,
                        &temp_status))
425  7            {
426  8    deactivate trail queue for trailid %d, cannot continue.", TrailID);
427  8                (void)rbe_log_stats(0,
428  8                  "Internal error: Cannot
429  8                return -1;
430  7            }
```

```
431  7
432  7        /* This test is to determine,
               * if a trail restore finished,
               * there may be another not yet started trail, if we
               * drive available the next trail restore will be
                 started.
               */
433  7
434  8

435  7
437  7        if ((0 != TrailRestoresLeft) &&
                   (HighestActiveTrail < TrailRestoresTotal) &&
                   (CountDrivesInUse < CountDrivesAvailable))
438  7        {
439  8            (void)rbe_user_error(0,
                   "Internal error: Cannot
441  8            HighestActiveTrail++;
443  8            if(0 != ActivateTrailQueue(HighestActiveTrail,
                          1,
                          &temp_status))
444  8
445  8
446  8            {
447  8                (void)rbe_user_error(0,
                       "Internal error: Cannot continue.",
448  8                return -1;
449  8            }

         activate trail queue(2) for trailid %d,

451  8            return -1;
452  8        }
453  8            if(0 != SetDrivesAcquired(
                          HighestActiveTrail, 1, &temp_status))
455  8
456  9            {
457  9                (void)rbe_user_error(0,
458  9
459  9                return -1;
460  9            }
         drive acquired(2) for trailid %d, cannot continue.",

461  9        if (0 > (temp_status = RunWorkItemRestoresForTrail(
                      HighestActiveTrail,
462  9                CountDrivesAvailable,
463  9                CancelRestoresReset,
464  9                &QuitFlag,
465  9                &CountDrivesInUse)))
466  9
467  9        {
468  8        /* runWorkItemRestoresForTrail does its own logging.
469  8         */
470  8            return -1;
471  8        }
472  9        if(temp_status == 0)
473  9        {
474  9            /* If this Trail had no work items we
475  9             * Should attempt to run the next trails
476  9             * work items. This would be an abnormal
477  9             * error if a trail queue had no work item
478  9             * restores.
479  9             */
480  9    restore had no work item to run().", HighestActiveTrail);
                 (void)rbe_log_stats(0,
                   "Internal error: Trail %d
481  9            return -1;
482  9        }
483  9        if(temp_status > 0)
             {
```

```
484  9                                   /* If at least on work item was started for this
485  9                                                                                    trail.
486  9                                    * then we have started a new trail.
487  8                                    */
488  8                                   TrailRestoresRunning++;
489  7
491  6                               }
492  5                           }
493  5                       } /* end for() */
494  4
495  2                   } /* else No Available fds */
496  2
498  2                   /*
499  2                    * ---------------------------------------
500  2                    * Terminate the loop if either
501  2                    * ---------------------------------------
502  2                    * 1] Sent the work items the quit. AND
503  2                    *    No Trail restores a running.
504  2                    *    OR
505  2                    * 2] No more Trail restores are left.
507  2                    */
508  2                   if(((0 == TrailRestoresRunning) && (SentQuit)) ||
509  1                      (0 == TrailRestoresLeft))
510  1                   {
511  1                       break;
512  1                   }
513  1               } /* end while(1) */
514  1               if((0 == TrailRestoresRunning) && (SentQuit))
515  1               {
516  2                   (void)rtbe_log_stats(0,
517  2                                    "Restore was quit by user.
518  1                                    Work item restore quit.");
519  1               }
520  1               return 0;
```

```
522          /* Functions needed
523          SendRunningWorkItemsQuit();
524          InterpretWorkItemRestoreResults();
525          */

527          static int
528          Select(int nfds,
529                 fd_set *readfds,
530                 fd_set *writefds,
531                 fd_set *exceptfds,
532                 struct timeval *timeout)
533          {
534  1           int retSelect;

536  1           do
537  1           {
538  2               retSelect = select(nfds,
539  2                            readfds,
540  2                            writefds,
541  2                            exceptfds,
542  2                            timeout);
544  1           } while ((-1 == retSelect) && (EINTR == errno));

546  1           return retSelect;
547          }
```

```c
eperrno
InitiateWorkItemRestore(const int SubmitObjID,
                        const int SubmitElemID)
{
    struct auxproc AuxprocVitals;
    eerrno_ry StartupAPResults = EXIT_FAILURE;
    int tempStatus;
    char *junk_executable[1024];
    char **junk_argv;
    char **AP_env = NULL;
    time_t StartTime;
    char clientName[256] = "";
    int status;
    time_t EndTime;
    int clientPort;

    /*
     * Lets see if there are environment variables to set.
     * The restore of the output variables are ignored.
     */
    if(E_SUCCESS != GetSOExecutionPhase(SubmitObjID,
                                        &junk_executable, 1024,
                                        &junk_argv,
                                        &AP_env,
                                        &SOStatus))
    {
        (void)the_user_error(0,
            "Internal Error: Could not get environment\n"
            variables.");

        return -1;
    }

    if (E_SUCCESS == GetSRsvcConnect(SubmitElemID,
                                     clientName, 256,
                                     &clientPort,
                                     &SOStatus))
    {
        StartupAPResults = StartupAuxprocess(0 /* XXX */,
                                             &AuxprocVitals,
                                             AP_env,
                                             clientName,
                                             clientPort);
    }
    else
    {
        (void)the_user_error(0,
            "Internal Error: Could not get Remote Client name &"
            "port to connect.");

        return -1;
    }

    if(E_SUCCESS != StartupAPResults)
    {
        /* StartupAuxprocess does its own logging */
        return -1;
    }

    /*
     * We need to close the bulk fd. This file descriptor
     * is not being used any more. If we do not close it
     * here we will have a file descriptor leak because
     * we won't be able to determine what it was when the
     * work item completes.
```

```c
     */
    close(AuxprocVitals.xp_fd_bulk_to_X);

    time(&StartTime);

    if(0 == newHandleSet())
    {
        (void)the_user_error(
            0, "Internal Error: Could not register handle set.");

        return -1;
    }

    if(0 > StartWorkItemRestore(rcp,
                                &AuxprocVitals,
                                SubmitObjID,
                                SubmitElemID))
    {
        /*
         * StartWorkItemRestore does logging if initialization fails
         */
        (void)the_user_error(
            0, "Error in StartWorkItemRestore SubmitObjID %d.",
            SubmitElemID);

        return -1;
    }

    /*
     * the following code kills auxproc when recv or xoplo do not
     *                                                          start
     * we do not want an auxproc sitting around.
     * If errors occur in deleteHandleSet or KillWorkItemRestore the
     *                                                          messages are
     * logged in those calls, plus
     * we already know there was an error and that
     * is why we are doing this right here.
     */
    time(&EndTime);

    deleteHandleSet();
    AuxprocVitals.xp_fd_from_x, EndTime, EP_KA_RECOVER_ALLFAIL, &status);

    KillWorkItemRestore(
        AuxprocVitals.xp_pid, AuxprocVitals.xp_fd_to_X);

    return 0;

}  /* InitiateWorkItemRestore() */
```

```c
    /*
     *
     *   interprets return.
     *   Drain progress.
     *   Send final progress for work item.
     *   Delete the handle set.
     */

static int
HandleWorkItemRestoreResults(int FromFD,
                             int *TrailID,
                             wi_restore_results *results)
{
    int  ret = 0;
    int  retries = 0;
    int  GetAuxprocResultsStatus;
    int  TempStatus;
    int  DrainResult;
    int  DrainedFD;
    int  wiCount;
    int  Auxprocpid;
    time_t EndTime;
    unsigned long jobstat;
    int  timeout = 3;  /* Lets try 3 seconds */
    boolean_t fromFDHangUp = FALSE;

    ToFD = getFromFD = ProgressFD = -1;

    while(! (fromFDHangUp))
    {
        GetAuxprocResultsStatus = GetAuxprocResults(FromFD, results);

        if(-1 == GetAuxprocResultsStatus)

        /* GetAuxprocResults() does its own logging */
        {
            (void)the_user_error(0," Error in GetAuxprocResults");
            return -1;
        }

        if(0 == GetAuxprocResultsStatus)
        {
            if(test_fd_hup(fromFD) == 1)
            {
                fromFDHangUp = TRUE;
            }

            /* The remote result are not always going to
             * be set. For example if the remote command
             * is not started correctly.
             */

            if(results->local_exit_set == TRUE)
            {
                break;
            }
            else
            {
                sleep(1);
                test_fd(fromFD);
                continue;
            }
        }
```

```c
        time(&EndTime);
    }

    return -1;

    if(0 != PushDrainRequest(FromFD, &TempStatus))
    {
        (void)the_user_error(0,
            "Internal error: Could not push drain "
            "request, cannot continue.");
    }

    if(0 != FindTrailQueueOfEnt(FromFD, TrailID, &TempStatus))
    {
        (void)the_user_error(0,
            "Internal error: Could not find trail id for "
            "finished work item, cannot continue.");
    }

    return -1;

    /* Lets give the progress thread a chance to drain keeping busy in
     * the meanwhile.
     */

    if(0 != DecrementRunningRef(*TrailID, &wiCount, &TempStatus))
    {
        (void)the_user_error(0,
            "Internal error: Could not decrement running "
            "work items for trail, cannot continue.");
    }

    return -1;

    if(0 != getPid(FromFD, &Auxprocpid, &TempStatus))
    {
        (void)the_user_error(0,
            "Internal error: Could not get auxproc pid "
            "for work item, cannot continue.");
    }

    return -1;

    if(0 != getHandleSet(
            FromFD, &ToFD, &getFromFD, &ProgressFD, &TempStatus))
    {
        (void)the_user_error(0,
            "Internal error: Could not get auxproc file "
            "descriptors for work item, cannot continue.");
    }

    return -1;

    if(FromFD != getFromFD)
    {
        (void)the_user_error(0,
            "Internal error: mismatch on from file "
            "descriptors for work item, cannot continue.");
    }

    while (0 != (ret = PopDrainResult(DrainedFD,
                    &TempFD,
                    &DrainResult,
                    &DrainedFD,
                    &TempStatus) && retries < 3))
    {
        retries++;
    }

    if (ret != 0)
    {
```

```
786 2          (void)ziho_user_error (0,
787 2              "internal error: Could not pop drain results,
                        cannot continue.");
789 2          }
790 2      )
791 2      return -1;
792 2      }
793 1
794 1      /*
795 1       * Send final Progress for work item. XXX
796 1       *
797 1       */
798 1
799 1      /* Translate the local and remote error statuses
800 1       * to an operrno value! */
801 1      switch (results->local_exit_status)
802 2      {
803 2      case XG_EXIT_ALLFAIL:
804 2          jobstat = EP_RB_RECOVER_ALLFAIL;
805 2          break;
806 2      case XG_EXIT_MANYFAIL:
807 2          jobstat = EP_RB_RECOVER_MANYFAIL;
808 2          break;
809 2      case XG_EXIT_FEWFAIL:
810 2          jobstat = EP_RB_RECOVER_FEWFAIL;
811 2          break;
812 2      case SPEXIT_REMOTE_STDERR_FAIL:
813 2      case SPEXIT_REMOTE_STDERR_PROTOCOL:
814 2          jobstat = EP_RB_RECOVER_CLIENT_STDERR_FAIL;
815 2          break;
816 2      case XG_EXIT_STOPPED:           /* treat like signal */
817 2      case XG_EXIT_SIGNAL:
818 2          /* check for signal termination vs all generic
                        failures */
819 2          if ( XG_EXIT_SIGNAL < results->local_exit_status ||
820 2               XG_EXIT_STOPPED == results->local_exit_status
821 2          /* killed by signal or stopped: separate error for sigfile */
822 2          if [XG_EXIT_SIGNAL & SIGFIPE == results->local_exit_status
823 2              jobstat = EP_RB_RECOVER_SIGFIPE;
824 2          else
825 2              jobstat = EP_RB_RECOVER_SERVER_SIGNAL;
826 2          }
827 2          else
828 2          { /* generic server failure, unless client failed too */
829 2              jobstat = EP_RB_RECOVER_SERVERFAIL;
830 2              if (0 != results->remote_exit_status)
831 2                  jobstat = EP_RB_RECOVER_BOTHFAIL;
832 2          }
833 2      }
834 2      else if(0 != results->remote_exit_status)
835 2          jobstat = EP_RB_RECOVER_CLIENTFAIL;
836 2      else
837 2          jobstat = E_SUCCESS;
838 2
839 2      if (0 != results->local_exit_status ||
840 2          (0 != results->remote_exit_status) ) )
841 2      {
842 2          int status=0;
843 2          int rc=0;
844 2          char *templateName=NULL;
845 2          char *winame=NULL;
846 2          char *trailsetName=NULL;
847 2
848 2          rc = getHandleSetInformation(fromFD,
                        &templateName,
849 2                   &winame,
```

```
849 2                   &trailsetName,
                        &status);
851 1
852 2          rbe_log_state(0, "Restore failure of \
853 2              "top level object: %s, template %s.",
854 2              STR_SURE(winame),
855 2              STR_SURE(templateName));
856 2
857 2          free(templateName);
858 2          free(winame);
859 2          free(trailsetName);
860 1
861 2      }
862 2      if ( 0 != deleteHandleSet(fromFD, jobstat, &TempStatus))
863 2      {
864 2          (void)ziho_user_error(0,
865 2              "internal error: Could not delete Handle
866                  Set, cannot continue.");
867 2          return -1;
868 2      }
869 2      if (0 != KillWorkItemRestore(AuxProcFd,
870 2              -1   /* Hack this arg is not needed yet */
871                  cmd_io ) )
872 2      {
873 2          (void)the_user_error(0,
874 2              "internal error: Could not kill finished
875                  auxproc, cannot continue.");
876 2          return -1;
877 2      }
878 2
879 2      close(toFD);
880 2      close(fromFD);
881 2      close(ProgressFD);
882 2
883 2      if(debugmode)
884 2      {
885 2          (void)the_user_error(0,
886 2              "DEBUG: HandleWorkItemRestoreResults AuxProc
887 2              PID %d just finished for trailid %d work items left = %d",
888 2              AuxProcFid,
889 2              *Trailid,
890 2              wlcount);
891 2      }
892 2
893 2      (void)the_user_error(0,
894 2          "DEBUG: HandleWorkItemRestoreResults Auxproc(
895 2          PID %d) results are local: %d  set!%s remote: %d set:%s.",
896 2          AuxProcFid,
897 2          results -> local_exit_status,
898 2          results -> local_exit_set ? "TRUE": "FALSE" ,
899 2          results -> remote_exit_status,
900 2          results -> remote_exit_set ? "TRUE": "FALSE");
889 1      return 0;
890 }
895 }  /* End HandleWorkItemRestoreResults() */
```

```
906   /*
907    * RunWorkItemRestoresForTrail()
908    * This function starts all the work item for the
909    * trail. For no this is set to one but concurrency
910    * will can be supported.
911    *
912    * Args:
913    *   (I)  trailID -- The id for this trail.
914    *   (I)  CountDrivesAvailable -- the total drives available to restore.
915    *   (O)  QuitFlag -- indicate whether the user has quit the restore.
916    *   (O)  CountDrivesInUse -- The count of trails in use by restore.
917    *
918    * Return int
919    *   if - if then an error has occurred.
920    *   if 0 or greater then the total drives started will be
921    *        returned.
922    */
923   static int
924   RunWorkItemRestoresForTrail(const int TrailID,
925                   const int CountDrivesAvailable,
926                   boolean_ty (*CancelRestoreForTrail)(),
927                   boolean_ty *QuitFlag,
928                   int *CountDrivesInUse)
929   {
930       int DrivesAcquiredForTrail;
931       int DriveConcurrencyForTrail;
932       int submitObjID;
933       int submitElementID;
934       int popResults = 0;
935       int temp_status;
936       int CountOfWorkItemRestoresStarted = 0;
937       int wlCount;
938
940       while(1)
941       {
944           if(0 != (popResults = PopWIFromTrailQueue(TrailID,
945                                   &submitObjID,
946                                   &submitElementID,
947                                   &temp_status)) &&
948               (SCHED_NO_MORE_JOBS != temp_status))
949           {
950               (void)the_user_error(0,
951                   "Internal error: Cannot pop work item off
952                   trail queue, cannot continue.");
953
954               return -1;
955           }
956
957           if((-1 == popResults) && (SCHED_NO_MORE_JOBS == temp_status))
958           {
959               return CountOfWorkItemRestoresStarted;
960           }
961
962           temp_status = InitiateWorkItemRestore(
963                           submitObjID, submitElementID);
964
966           if(temp_status != 0)
```

```
964           {
965               /* InitiateWorkItemRestore() does its own logging */
966               (void)the_user_error(0, "Error in InitiateWorkItemRestore,"
967                   "submitObjID %d, submitElementID %d", submitObjID,
968                   submitElementID);
969
970               return -1;
971           }
972
973           if(0 != IncrementRunningWI(TrailID, &wlCount, &temp_status))
974           {
975               (void)the_user_error(0,
976                   "Internal error: Could not increment
977                   running work items for trail, cannot continue.");
978
979               return -1;
980           }
981
982           CountOfWorkItemRestoresStarted++;
983
984           if(0 != DriveAcquiredForTrail(TrailID,
985                       &DriveAcquiredForTrail,
986                       &temp_status))
987           {
988               (void)the_user_error(0,
989                   "Internal error: Cannot get drives
990                   acquired, cannot continue.");
991
992               return -1;
993           }
994
995           if(0 != GetDriveConcurrency(TrailID,
996                       &DriveConcurrencyForTrail,
997                       &temp_status))
998           {
999               (void)the_user_error(0,
1000                  "Internal error: Cannot get drive
1001                  concurrency, cannot continue.");
1002
1003              return -1;
1004          }
1005
1006          *QuitFlag = CancelRestoreForTrail();
1007
1008          if((DriveAcquiredForTrail < DriveConcurrencyForTrail) &&
1009             (*CountDrivesAvailable < CountDrivesInUse) &&
1010             (FALSE == *QuitFlag))
1011          {
1012              continue;
1013          }
1014          else
              {
                  break;
              }
          }
          return CountOfWorkItemRestoresStarted;
      }  /* RunWorkItemRestoresForTrail() */
```

```
1018      /* Stub */
1019      static int DetermineGlobalDriveUse()
1020  1   {
1021  1      /* Limiting to MAXINT === not limiting... Need resource management
1022  1       * to do this properly.
               NOTE: This should now work like sb_dc_restore does.
             */
1023  1      return MAXINT;
1024  1   }
1025  1
```

```
1028      static int
1029      SendRunningWorkItemsQuit()
1030  1   {
1031  1      int *APList;
1032  1      int count;
1033  1      int status;
1034  1      int index;
1035
1036  1      if(0 != getPIDList(&count, &APList, &status))
1037  2      {
1038  2         (void)the_user_error(0, "Internal error: Cannot get auxproc pid list,
                                         cannot continue.");
1039  2
1040  2         return -1;
1041  2
             }
1043  1      for(index = 0; index < count; index++)
1044  2      {
1045  2         QuitWorkItemRestore(APList[index]);
1046  2
1047  1      }
1048  1      return 0;
             }
```

```
1050        /*
1051         * Stub this out for now.
1052         */
1053    static int
1054    InterpretWorkItemRestoreResults(wi_restore_results *results)
1055    {
1056        return 0;
1057    }
```

```
1059    static void
1060    DebugLogFds(char *error_msg,
1061                fd_set *fds)
1062    {
1063        int index, fd_count = 0;
1064        char buffer[4096];
1065        char *bufptr = (char *)buffer;

1068        for(index=0;
1069            index < 1024;
1070            index++)
1071        {
1072            if( FD_ISSET(index, fds))
1073            {
1074                int size = 0;
1075                size = sprintf(bufptr, "%d,", index);
1076                bufptr += size;
1077                fd_count++;
1078            }
1079        }
1080        rho_log_stats(0, "%s fd_count: %d :: {%s}\n",
1081                error_msg, fd_count, buffer);
1082    }
```

```
1085    static int
1086    test_fd(int fd)
1087 1  {   fd_set read_fd;
1088 1      int ret_select;
1089 1      int ret_select;
1090 1      struct timeval timeout = {0, 0};
1092 1      FD_ZERO(&read_fd);
1094 1      FD_SET(fd, &read_fd);
1096 1      do
1097 2      {
1098 2          ret_select = select(fd + 1, &read_fd, NULL, NULL, &timeout);
1100 1      } while((-1 == ret_select) && (EINTR == errno));
1102 1      return ret_select;
1104    }
```

```
1106    /*
1107     *  test_fd_hup ()
1108     *
1109     *  Description: Test the supplied file descriptor to see if
1110     *  it has had the hang up condition.
1111     *
1112     *  Args:
1113     *  Input fd - the file descriptor to check for the hang up condition.
1114     *
1115     *  Returns:
1116     *  1 for HUP event received on fd.
1117     *  0 No HUP event received on fd.
1118     *  -1 errno set.
1119     *
1120     */
1121    static int
1122    test_fd_hup(int fd)
1123 1  {   struct pollfd fds;
1124 1      int ret_poll;
1126 1      if(fd < 0)
1127 2      {
1128 2          errno = EINVAL;
1129 2          return -1;
1130 2      }
1132 1      fds.fd = fd;
1133 1      fds.events = POLLIN;
1134 1      fds.revents = 0;   /* initialize */
1136 1      do
1137 2      {
1138 2          ret_poll = poll(&fds, 1, 0);
1139 1      } while((-1 == ret_poll) && (EINVAL == errno));
1141 1      if(POLLHUP & fds.revents)
1142 2      {
1143 2          return 1;
1144 2      }
1145 1      if(-1 == ret_poll)
1146 2      {
1147 2          return -1;
1148 2      }
1149 1      else
1150 2      {
1151 2          return 0;
1152 2      }
1154    } /* end test_fd_hup() */
```

```c
1    /************************************************************
2    **
3    ** File Name:    RSLauxmgr.c
4    **
5    ** Copyright (c) 1998,1999 by EMC Corporation.
6    **
7    ** Purpose:
8    **    -- The intent of the contents of this file is to implement the
9    **       functions the control execution of auxproc or the
10   **       running of auxproc.
11   **
12   ** Library:
13   **
14   **    These functions are provided to allow:
15   **       - The piping, fork, dupping coding and exec of auxproc.
16   **       - The starting of work item restores.
17   **       - The quitting of work item restores.
18   **       - The getting results of work item restores.
19   **
20   **    The following functions comprise restoral management:
21   **
22   **
23   ** Compile-Time Options:
24   **    This section must list any compile time definitions
25   **    which will affect this header.
26   **
27   **
28   **
29   ** Feature test switches:
30   **    Standard defines required to turn on OS features go here.
31   **
32   **    The following is required for code that uses POSIX API's.
33   **    Remove for non-POSIX, non-portable code.
34   **/
35
36   #define _POSIX_SOURCE 1
37
38   /*
39   ** System headers.
40   */
41   #include <sys/wait.h>
42   #include <sys/types.h>
43   #include <unistd.h>
44   #include <string.h>
45   #include <stdlib.h>
46
47   /*
48   ** Epoch headers.
49   */
50   #include <eb/eb_port.h>
51   #include <eb/lb_log.h>
52   #include <eb/util/eb_normalize.h>
53   #include <eb/util/mutil.h>
54   #include <abreport/avl.h>
```

```c
65   /*
66   ** Local headers
67   */
68   #include <RSLintern.h>
69   #include <RSLbrmain.h>
70   #include <restore/RDNRESubmitApi.h>
71   #include <RSLiSubmit.h>
72   #include <RSLauxSupp.h>
73
74   #define SUBMIT_FIELD_MAX 2048
75
76   extern int putcwv(const char *string);
77
78   extern char *strsignal(int sig);
79
80   extern int ChildDone(int child_pid, int *child_result);
81
82   static void
83   reset_recovery_privileges(struct recover_context *rcx,
84                             int changed);
85
86   static void
87   set_recovery_privileges(struct recover_context *rcx,
88                           int *changed);
89
90   static char *
91   generate_rcmpath(int SubmitObjectId, int SubmitElemId);
92
93   static char *
94   StartupAuxprocess()
95   /*
96   ** Description:
97   **    This function is on pipe, fork, & exec auxproc. After which it tests
98   **    its the just started auxproc with the Ping command. Some additional
99   **    processing is it closes the fds that are not associated with
100  **    auxproc.
101  **
102  **    but are inherited from the parent (
103  **                         restore engine). if any environment
104  **    variable need to be set for auxproc, they are set.
105  **
106  ** Args:
107  **    (inp)  debugmode -- int 1 for debug and 0 for no debug.
108  **    (out)  struct auxproc *xp -- preallocated structure to return vital
109  **                                 info
110  **    (char) *auxproc_envp --  environment variables to be appended to
111  **                             auxproc's
112  **                             environment;
113  **                             rest of the environment is
114  **                             inherited from auxproc.
115  **                             Format is "ENV=value\0"
116  **    int    client --  used for the client initiated restore this is the
117  **                         client name
118  **    char   *socketClientAm -- Used to identify the client
119  **                              contact the client
120  **    int    clientSocketPort --  client port number on which to
121  **                                 contact the client
```

Notes:
```
**    Notes:
**       auxproc_envp can be NULL indicating no environment to append to
**                         auxproc.
**       This is a char ** which last char * should be NULL.
**
**    Inherited from:
```

```
121   *    static eerrno_ty   setup_aux_process(struct recover_context *rcx)
122   */
123  {
125   eerrno_ty
126   StartupAuxprocess(struct            debugmode,
127                     int              *op,
128                     char             *auxproc_envp,
129                     char             *socketClientbn,
130                     int              clientSocketport)
131  {
133   /*
134    *
135    * NOTES: I really want to be reliant on the recover_context struct.
136    *
137    * 1) I need to fork()/ auxproc
138    * 2) I need to fork()/ exec()/ auxproc
139    * 3) Is fork I really really going to work.
140    */
141   #define RFD        0       /* in a pipe, fd 0 is the read descriptor */
142   #define WFD        1       /* and fd 1 is the write descriptor */
143       int save_errno;
144       int fd;
145       char *resultsbuf = NULL;
146       char *auxproc_pathname = AUXPROCPATHNAME;
147       char *auxproc_executable = AUXPROCNAME;
149       int ping_status;
150       int auxproc_index = 0;   /* alcon Remove */
151       int cmd_pipe_to[2];
152       int cmd_pipe_from[2];
153       int bulk_pipe_to[2];
154       int prog_pipe_from[2];
156       if (pipe(cmd_pipe_to) == -1 ||
157           pipe(cmd_pipe_from) == -1 ||
158           pipe(bulk_pipe_to) == -1 ||
159           pipe(prog_pipe_from) == -1)
160       {
161           save_errno = errno;
162           rbe_log_stats(RBRECOVER_MKERR(errno), "pipe() failed");
163           return(RBRECOVER_MKERR(save_errno));
164       }
165       /*
166        * This below appends environment variables to
167        * the environment that auxproc inherits from the
168        * restore engine.
169        */
171       if(NULL != auxproc_envp)
172       {
173           int index;
174           for(index=0; NULL != auxproc_envp[index]; index++)
175           {
176               if(0 != putenv(auxproc_envp[index]))
177               {
178                   rbe_log_stats(RBRECOVER_MKERR(errno),
179                       "Unable to set auxproc environment %s, "
180                       "for auxproc PID %d.",
181                       auxproc_envp[index], getpid());
182               }
183           }
184       }
186  }
```

```
188       switch (op->xp_pid = EDK(fork))
189       {
190       case -1:  /* Error */
191           save_errno = errno;
192           rbe_log_stats(RBRECOVER_MKERR(errno));
193           return(RBRECOVER_MKERR(save_errno));
195       case 0:  /* child */
196       {
198           char procnum_str[32];
199           char r_fd_str[32];
200           char w_fd_str[32];
201           char r_bulk_fd_str[32];
202           char w_prog_fd_str[32];
203           char debugmode_str[32];
204           char socket_host_str[255];
205           char socket_port_str[32];   /* Not sure what this should be */
207           socket_port_str[0] = '\0';
210           (void) sprintf(procnum_str, "%d", auxproc_index);
211           (void) sprintf(r_fd_str, "%d", cmd_pipe_to[RFD]);
212           (void) sprintf(w_fd_str, "%d", cmd_pipe_from[WFD]);
213           (void) sprintf(r_bulk_fd_str, "%d", bulk_pipe_to[RFD]);
214           (void) sprintf(w_prog_fd_str, "%d", prog_pipe_from[WFD]);
215           (void) sprintf(debugmode_str, "%d", debugmode);
216           if ( NULL != socketClientbn)
217           {
218               (void) sprintf(socket_host_str, "%s", socketClientbn);
219               (void) sprintf(socket_port_str, "%d", clientSocketport);
220           }
223           else
224           {
224               socket_host_str[0] = 0;
                  socket_port_str[0] = 0;
             }
227           ret_exec = execlp(
228               Auxproc_pathname,
                  Auxproc_executable,         /* prog to execute */
                  procnum_str,
                  r_fd_str,
                  w_fd_str,
                  r_bulk_fd_str,
                  w_prog_fd_str,
                  debugmode_str,
                  socket_host_str,
                  socket_port_str,
                  (char *)0);
             rbe_log_stats(RBRECOVER_MKERR(errno),
                  "Unable to exec %s, for %s PID %d.",
                  AUXPROCNAME,
                  AUXPROCNAME,
                  getpid());
239           _exit (1);
243       default:  /* parent */
246           _exit (1);
247       }
248   }
```

```
        /*
         * The parent has no need for the fd
         * used to write "to" the parent, nor
         * the fds used to read "from" the parent.
         * If these are not close FPPE and SIGPIPE
         * may be missed by the writer when the
         * intended reader dies.
         */

        (void)close(cmd_pipe_to[RFD]);
        (void)close(bulk_pipe_to[RFD]);
        (void)close(cmd_pipe_from[WFD]);
        (void)close(prog_pipe_from[WFD]);

        /*
         * but does want to save the other fds
         */

        xp->xp_fd_to_x      = cmd_pipe_to[WFD];
        xp->xp_bulk_fd_to_x = bulk_pipe_to[WFD];
        xp->xp_fd_from_x    = cmd_pipe_from[RFD];
        xp->xp_prog_from_x  = prog_pipe_from[RFD];

        /*
         * In debugmode, exec the separate-process
         * version of the auxproc (ptrace issue)
         */

        if (debugmode)
        {
            auxmsgpacket(xp->xp_fd_to_x, 'X', 0, "");
        }

#define PING_TEST_STR      "abc"
#define PING_TEST_STR_SIZE 4  /* include the '\0' */

        auxmsgpacket(fd, 'P', PING_TEST_STR_SIZE, PING_TEST_STR);

        fd = xp->xp_fd_from_x;

        ping_status = auxresults(fd, 'P', 0, &resultsbuf);

        if ((-1 == ping_status) ||
            (NULL == resultsbuf) ||
            (strcmp(resultsbuf, PING_TEST_STR) != 0))
        {
            rec_epri_log_cmn(SIBL_CSE_NO_PING_AUXPROC, NULL);
            rbe_log_state.s0 != ping start-up;
            return(BF_RA_RECOVER_AUXPROC_DIED);
        }

        free(resultsbuf);

        return(E_SUCCESS);
    }
#undef PING_TEST_STR
#undef PING_TEST_STR_SIZE
#undef RFD
#undef WFD
#undef MAX_FD

/* end of setup_aux_process() */
```

```
/****************************************************************
 *
 * start_cpolgen()
 *
 *     This function initiates a work item restore. It first determines
 *     the size of the restore command to send to auxproc, malloc's the
 *     memory and creates the restore command and sends it to auxproc.
 *     Auxproc will start the rcmd (if necessary) and start xcpligrpm.
 *     Auxproc will send the initialization reply which is read by this
 *     function. The results are for initialization.
 *
 * Args:
 *     (I)   auxprocnum     -- auxproc number (may become obsolete.
 *     (I)   xp             -- auxproc context pointer (limited use)
 *     (I)   rcx            -- Recover Context struct (limited use)
 *     (I)   submitobjectID -- identifies what and how to run restore
 *     (O)   results        -- Of the work item restore initialization.
 *     (O)   err_str        -- Of work item restore initialization.
 *
 * Returns:
 *     xcpligen's pid for success, -1 for failure.
 *
 * NOTES: The recover context structure is carefully used below.
 *     rcx -- The recover context structure should be used only to get the global values
 *     xp  -- The auxproc structure should be used only to get the config structure.
 *            like the xcpligen executable name and the config structure.
 *
 *     Submit Object should be used to determine user id,
 *     admin privileges, and other values that would not vary
 *     for a potentially multi work item restore.
 *
 *     Submit Element should be used to determine anything that
 *     could be potentially unique for a work item restore.
 *
 *     rcmdinfo is the command line for the remote command.
 *
 ****************************************************************/

#define MAX_SUBMIT_FIELD 2048

int
start_cpolgen(struct recover_context *rcx,
              struct auxproc *xp,
              int submitobjectID,
              int auxprocnum,
              char *resultsbuf,
              char **results,
              char **err_str)
{
    char  *auxproc_databuf;
    size_t data_len = 0;
    int    i;
    int    retfd;

    /* For Initialization results */
    int               pid = -1;
    rcmd_pkto_info   *pktdp;
    rcmd_pkto_info    pkto_buffer;
    int               auxprocnum;
    char             *resultsbuf = NULL;
    rcmd_pkto_info   *resultsbufptr = NULL;
    char             *err_str;
    struct mark_summary submit,
    struct mark_summary summary;
```

```
379  1      /* Args for xcpiogen */
380  1          char *xcpiogen_argv0;
381  1          int  xcpiogen_argc;
382  1          char *submit_file_flag  = "-S";
383  1          char *output_file_flag  = "-fd";
384  1          char temp_submit_file[MAX_SUBMIT_FIELD];
385  1          u_hyper total_bytes;
386  1          char total_bytes_flag = 'B';
387  1          char *progress_report_flag = "-p";      /* From the summary */
388  1          char *bufsize_ptr = NULL;
389  1          char bufsize_buffer[20];
390
391  1      RBC_WORKGROUP *pg;
392  1      RBC_WORKITEM *pi;
393
394  1      /* temporary variable for the submit object / element fields. */
395  1          char temp_effective_uidname[MAX_SUBMIT_FIELD];
396  1          char temp_socket_host[MAX_SUBMIT_FIELD];
397  1          int  temp_socket_port;
398  1          char temp_work_item_name[MAX_SUBMIT_FIELD];
399  1          char temp_submit_file[MAX_SUBMIT_FIELD];
400  1          int  GetSSStatus = 0;
401  1          int  GetSSStatus = 0;
402
403  1      if( GetSBWorkItemName(submitobjectID, submitElementID,
404  2                  temp_work_item_name, MAX_SUBMIT_FIELD,
405  2                  &GetSSStatus) != 0)
406  1          return -1;
407
408  1      rbc_log_stats(0, "Unable to get work item name.");
409  1          return -1;
410
411  1      if( GetSBEffectiveUserName(submitobjectID,
412  2                  temp_effective_uidname,
413  2                  MAX_SUBMIT_FIELD, &GetSSStatus) != 0)
414  2
415  1          rbc_log_stats(0, "Unable to get user name.");
416  2          return -1;
417
418  1      if( GetSBRemoteConnect(submitobjectID, submitElementID,
419  2                  temp_socket_host,
420  2                  &temp_socket_port, &GetSSStatus) != 0)
421
422  1      rbc_log_stats(0, "Unable to get socket host/port pair.");
423  1          return -1;
424
425  1      if( GetSBSubmitFile(submitobjectID, submitElementID,
426  2                  temp_submit_file, MAX_SUBMIT_FIELD,
427  2                  &GetSSStatus) != 0)
428
429  1      rbc_log_stats(0, "Unable to get submit file.");
430  1          return -1;
431
432  1      /*
433  1       * .s for '\0' characters
434  1       */
435
436  1      data_len += strlen(rcmdinfo[0]) + 1;      /* rcmd-hostname */
437  1          data_len += strlen(rcmdinfo[1]) + 1;  /* rcmd-locuser */
438  1          data_len += strlen(rcmdinfo[2]) + 1;  /* rcmd-remuser */
439
```

```
444  1      data_len += strlen(rcmdinfo[3]) + 1;      /* rcmd-cmd */
445
446  1      data_len += strlen(temp_effective_uidname) + 1;
447
448  1      data_len += sizeof (int);
449  1          data_len += sizeof (int);
450  1          data_len += strlen( rc_cpiogen_executable) + 1;
451
452  1      data_len += sizeof (int);
453
454  1      else
455
456  1      xcpiogen_argv0 = strchr(rcx -> rc_cpiogen_executable, '/');
457  1          if (NULL != xcpiogen_argv0)
458
459  1          ++xcpiogen_argv0;
460
461  1      xcpiogen_argv0 = rcx -> rc_cpiogen_executable;
462  1          if (NULL != xcpiogen_argv0)
463  1          ++xcpiogen_argv0;
464
465  1      }
466  1      data_len += sizeof (int);
467
468  1      else
469
470  1      }
471  1      data_len += strlen(output_file_flag) + 1;      /* fd info */
472  1          data_len += strlen(temp_submit_file) + 1;  /* flags */
473
474  1      data_len += strlen(progress_report_flag) + 1;  /* xcpiogen-cmd */
475
```

```
444  1      data_len += strlen(temp_effective_uidname) + 1;      /* rcmd-cmd */
446  1      data_len += strlen(temp_effective_uidname) + 1;
448  1      data_len += sizeof (int);
449  1      data_len += sizeof (int);
450  1      data_len += strlen( rc_cpiogen_executable) + 1;      /* fd info */
451
                                                               /* flags */
                                                               /* xcpiogen-cmd */
                                                               /* xcpiogen argv[0] */
                                                               /* xcpiogen argv[1] */
                                                               /* xcpiogen argv[2] */
                                                               /* xcpiogen argv[3] */
                                                               /* xcpiogen argv[4] */

476  1      xcpiogen_argc = 7;
477
478  1      /*
479  2       * Get the current total number of bytes to be processed from
480  2       * the mark summary so u_hyper so that we can then convert it to
481  2       * a decimal string to be passed to xcpiogen().
482  2       */
483
484  1      if(0 != GetSBMarkedSummary(submitobjectID,
485  2                  submit_summary,
486  2                  &submit_summary,
487  2                  &GetSSStatus))
488  2
489  1      /* This is not a critical error. This may cause progress
490  2       * reporting problems!
491  2       */
492  1      rbc_log_stats(0, "Unable to set submit size for xcpiogen.");
493  1          total_bytes = u_lto_uint(0);
494
495      else
496
497  1      total_bytes = submit_summary.len_mod_files;
498
499  1      /*
500  2       * Add the size of the "total_bytes" flag and value string
501  2       */
502  1      total_bytes_string = u_hyper_to_decimal(total_bytes);
503  1          data_len += strlen(total_bytes_string);
```

```
504 1          */
506 1          data_len += strlen(total_bytes_flag) + 1;
507 2          data_len += strlen(rcx->rc_config_socket_host);   /* xpiogen argv[5] */
                                                                 /* 1 */
                                                                 /* xpiogen argv[6] */
509 1          /*
510 1           * Add size for db API socket info
511 1           */
513 1          data_len += sizeof (int);
514 1          data_len += strlen(temp_socket_host) + 1;    /* socket port # */

516 1          /*
517 1           * locate work item in config info & get length of filespec
518 1           * %%% changed back in inner loop to 'goto' to resume with
519 1           * found item.
520 1           */
522 1          for (pi = NULL, pg = rcx->rc_config->rc_grouplist;   /* OK */
523 1               NULL != pg;
                    pg = pg->next)
525 2          {
527 2              for (pi = pg->pwdlist; NULL != pi; pi = pi->next)
529 2              {
530 4                  if (0 == strcmp(pi->name, temp_workitem_name))
531 4                      goto stopsearch;      /* %%% exit both loops */
533 3                  else
535 3                      data_len += strlen(pi->list)+1;
536 3              }
537 2          }
539 1      stopsearch:
540 1          if (pi != NULL)
541 2              pg = NULL;
543 1
545 1          if (NULL != pi && NULL != pg) {   /* pi->recover_server_bufsize */
546 2              data_len += strlen(temp_workitem_name) + 1;
548 1              sprintf(buffsize_buffer, "-R%d", pi->recover_server_bufsize);
549 2              data_len += strlen(buffsize_buffer);
550 2              data_len += strlen(buffsize_buffer);
551 1              xpiogen_argc++;        /* one more arg to xpiogen */
553 1          }
555 1          /*
556 1           * Allocate memory to hold all this gunk we need to
557 2           * above towards our auxiliary processes
558 1           */
560 1          auxproc_databuf = sm_fmalloc((unsigned)data_len);
562 1          /*
563 1           * Fill in the gunk.  First, the cmd info for the rsh part.
564 1           */
566 1          p = auxproc_databuf;
```

```
567 1          for (i = 0; i < 4; i++)
568 2          {
569 2              (void)strcpy(p, rcmdinfo[i]);
570 2              p += strlen(p)+1;
571 1          }
573 1          /*
574 1           * The human username which the remote should operate as
575 1           */
577 1          (void)strcpy(p, temp_effective_uidname);
578 1          p += strlen(p)+1;
580 1          /*
581 1           * The "xpiogen-cmd-fd-info", which tells auxproc which
582 1           * arg has the sprintf format string to insert the
583 1           * actual output fd number.  Remember: auxproc sets up
584 1           * the pipe between the xpiogen and the rcmd.  This
585 1           * below arg is sent to auxproc to tell auxproc which
586 1           * argument to update the format string with the outputfd.
587 1           */
589 1          i = 3;                          /* For argv[3] */
591 1          memcpy(p, &i, sizeof (int));
593 1          p += sizeof (int);
595 1          /*
596 1           * the flags, which are always zero currently
597 1           */
599 1          i = 0;
601 1          memcpy(p, &i, sizeof (int));
603 1          p += sizeof (int);
605 1          /*
606 1           * The "xpiogen command" that we will run locally.
607 1           */
608 1          (void)strcpy(p, rcx->rc_cpiogen_executable);
610 1          p += strlen(p)+1;
612 1          /*
613 1           * The argc for the "xpiogen command", and its argv vector.
614 1           */
615 1          memcpy(p, &xpiogen_argc, sizeof (int));
617 1          p += sizeof (int);
618 1          /*
619 1           * The argv vector, which is argv0, the outputfd
620 1           * thingy, and the progress report mode flag.
621 1           */
623 1          memcpy(p, &xpiogen_argv0, sizeof (int));
625 1          p += sizeof (int);
627 1          (void)strcpy(p, xpiogen_argv0);
628 1          p += strlen(p)+1;
630 1          (void)strcpy(p, temp_submit_file);   /* xpiogen argv[1] */
631 1          p += strlen(submit_file) + 1;
633 1          (void)strcpy(p, temp_submit_file);   /* xpiogen argv[2] */
639 1          p += strlen(temp_submit_file) + 1;
641 1          (void)strcpy(p, outputfd_func);      /* xpiogen argv[3] */
612 1          p += strlen(outputfd_func);
```

```c
616 1        (void)strcpy(p, progress_report_flag);  /* xcpiogen argv[4] */
615 1        p += strlen(progress_report_flag) + 1;

638 2        if (bufsizeP != NULL)
637 2        {
                (void)strcpy(p, bufsizeP);  /* xcpiogen argv[??] */
                p += strlen(bufsizeP) + 1;
641 2        }

             /*
              * Follow this with the total bytes flag and value.
              */
647 1        strcpy(p, total_bytes_flag);  /* xcpiogen argv[??] */
648 1        p += strlen(total_bytes_flag) + 1;
649 1        strcpy(p, total_bytes_stringP);  /* xcpiogen argv[??] */
650 1        p += strlen(total_bytes_stringP) + 1;

             /*
              * socket info for db API
              */
656 1        (void)memcpy(p, (char *)&temp_socket_port, sizeof (int));
657 1        p += sizeof (int);

             /*
              * socket host name for db API
              */
660 1        (void)strcpy(p, temp_socket_host);
661 1        p += strlen(p)+1;

664 2        if (NULL != pi)  /* send filespec */
665 2        {
                (void)strcpy(p, pi->list);
                p += strlen(p)+1;
669 2        }
670 1        else
671 2        {
                *p++ = 0;
674 2        }

             /*
              * workitem name
              */
677 1        (void)strcpy(p, temp_workitem_name);
681 1        p += strlen(p)+1;

             /*
              * assert that our arithmetic above was done correctly
              */
683 1        if ((size_t)(p - auxproc_databuf) != data_len)
688 2        {
                the_log_stats(0, "assertion failed: cmd size miscount");
                return -1;
691 2        }

             /*
              * Send the restore command to auxproc. Auxproc will start
              * The remote command (if necessary) and xcpiogen.
              */
694 1        auxmemcpacket(xp-> xp_fd_to_x,
```

```c
699 1        }

             /*
              * Obtain the fork status
              */
705 1        resfd = xp-> xp_fd_from_x;
707 1        i = auxresults(resfd, '0', 0, &resultsdbufP);

709 1        if (i < 0)
710 2        {
                the_log_stats(0,
                    "*** Error while starting auxproc for work item \"%s\"",
                    temp_workitem_name);
                null_free (resultsdbufP);
                return -1;
716 2        }

             /* This memory management is crap */
718 1        pktOp = &pktOp_buffer;
            memcpy(pktOp, resultsdbufP, sizeof(pktOp_buffer));
            memcpy(resultsd, resultsdbufP, sizeof(pktOp_buffer));

721 2        if (pktOp->msglen > 0)
724 2        {
                errstr = resultsdbufP + sizeof *pktOp;
                *err_str = esl_strdup(errstr);
728 2        }
            else
730 2        {
                *err_str = esl_strdup("");
733 2        }

             /*
              * If the fork failed, the cpiogen start fails
              */
735 2        if (0 != pktOp->failcode)
739 2        {
                int jnk;

                if (strlen(errstr) > (size_t)0)
                    the_log_stats(0,
                        "for work item \"%s\"", errstr,
                        temp_workitem_name);

                free (resultsdbufP);

                /*
                 * collect the useless 'r' reply packet
                 */
                resultsdbufP = (char *)&jnk;
                (void)auxresults(resfd, 'r', sizeof (int), &resultsdbufP);
                return -1;
```

```
762  1        /*
     1         * Caller assumes responsibility for (eventually)
     1         * collected exit status of remote and local programs.
     1         */
764  1
765  1
767  1        pid = pktop->pidi;

769  1        free (resultsbufptr);

771  1        return pid;
772  1    }   /* end of start_cpiogen() */
```

```
775  1        /*
776  1         * 1) Remove rcx references.
     1         */
779  1    static char *
780  1    make_remote_cpiogen_cmd(struct recover_context *rcx,
     1                            int SubmitObjectID,
     1                            int SubmitElemID)
782  1    {
783  1        char *rcmdpath = generate_rcmdpath(SubmitObjectID,
     1                                           SubmitElemID);
784  1        char *minus_c = "";
785  1        char *minus_c_arg = "";
787  1        char *noclobber = "";
788  1        char *rcmddebugflag = "";
789  1        char *cmd;
791  1        unsigned len;
792  1        RBC_WORKITEM *work_itemP;
793  1        char *sufsizeP = "";
794  1        char bufsize_buffer[20];

796  1        char temp_dirtop[SUBMIT_FIELD_MAX];
797  1        char temp_workitem_name[SUBMIT_FIELD_MAX];
798  1        OverwritePolicy temp_overwrite_policy;
799  1        int GetSESStatus = 0;

801  1        if (NULL == rcmdpath)
802  2        {
803  2            return NULL;
804  1        }

806  1        temp_overwrite_policy = GetSESGetOverwritePolicy(SubmitObjectID,
807  1                                                         SubmitElemID,
     1                                                         &GetSESStatus);

809  1        if (0 != GetSESStatus)
810  2        {
811  2            rbc_log_state(0, "Unable to get overwrite policy.");
812  2            return NULL;
813  1        }

815  1        if (GetSESdirTop(SubmitObjectID, SubmitElemID,
816  2                         temp_dirtop, SUBMIT_FIELD_MAX,
817  2                         &GetSESStatus) != 0)
818  1        {
819  2            rbc_log_state(0, "Unable to get dirtop.");
820  2            return NULL;
821  1        }

823  1        if (GetSESWorkItemName(SubmitObjectID, SubmitElemID,
824  2                               temp_workitem_name,
825  2                               SUBMIT_FIELD_MAX,
826  2                               &GetSESStatus) != 0)
827  1        {
828  2            rbc_log_state(0, "Unable to get work item name.");
829  2            return NULL;
830  1        }

832  1        if (temp_dirtop != NULL)
833  2        {
834  2            minus_c = " -c ";
835  2            minus_c_arg = temp_dirtop;
836  1        }

838  1        work_itemP = rbc_find_workitem_in_config(temp_workitem_name,
```

```
                        NULL,        NULL,
                        ECX->rc_config, /* OK */
                        RBC_FIND_WORKITEM_NO_OPTIONS);

        if (work_itemP != NULL
            && le DEFAULT_DB_BUFSIZE != work_itemP->recover_client_bufsize)
        {
            sprintf(bufsize_buffer, " -A -B %d',
                    work_itemP->recover_client_bufsize);
            bufsize = bufsize_buffer;
        }

        switch (temp_overwrite_policy)
        {
        case RC_OVERPOL_NO_CLOBBER:
            noclobber = " -A -noclobber";
            break;
        case RC_OVERPOL_NEW_CLOBBER:
            noclobber = " -A -onewer";
            break;
        default:
            break;       /* do something better here */
        }

        if (debugmode)
        {
            static char debugarg[100];

            (void)sprintf(debugarg, " -X %s",  getpid();  /* /tmp/RBRdebug%d */
                                                         );

            remdebugflag = debugarg;
        }

        len = strlen(cmdpath) +
              strlen(bufsize) +
              strlen(remdebugflag) +
              strlen(minus_c) +
              strlen(minus_c_arg) +
              strlen(noclobber) +
              1;       /* for '\0' */

        cmd = sm_fmalloc(len);
        (void)sprintf(cmd, "%s%s%s%s%s%s",
                      cmdpath,
                      bufsize,
                      remdebugflag,
                      minus_c,
                      minus_c_arg,
                      noclobber);

        return cmd;
    }       /* end of make_remote_cpiogen_cmd() */
```

```
    /*
     * 1) Remove rcx references.
     * 2) Research whether rbc_normalize updates.
     *
     * Construct the path name of the remote command that
     * will be executed on the destination client.
     * Return ptr to constructed string.
     * NOTE: caller must copy if string is to be preserved.
     */

    static char *
    generate_rcmdpath(int SubmitLObjectID,
                      int SubmitElemID)
    {
        static char *mybuf = NULL;
        size_t len_needed;
        char *p;
        char *pph;       /* pointer to %h */
        char *q;
        char *term_host;

        char temp_scriptname[SUBMTL_FIELD_MAX];
        char temp_client_hostname[SUBMTL_FIELD_MAX];

        int GetSIDStatus = 0;

        if (GetSIDandScriptname(SubmitLObjectID, SubmitElemID,
                                temp_scriptname, SUBMTL_FIELD_MAX,
                                &GetSIDStatus) != 0)
        {
            the_log_stats(0, "Unable to get rcmd script name.");
            return NULL;
        }

        if (GetSEDdestClientName(SubmitLObjectID, SubmitElemID,
                                 temp_client_hostname, SUBMTL_FIELD_MAX,
                                 &GetSEDStatus) != 0)
        {
            the_log_stats(0, "Unable to get client destination name.");
            return NULL;
        }

        /*
         * If there is a %h in the rc_client_scriptname,
         * that means put the hostname in there.
         * Sorry, no escapes implemented. You can't have an
         * rc_client_scriptname with a literal %h in it. Tough.
         */

        for (pph = temp_scriptname; *pph != '\0', pph++)
        {
            if (*pph == '%' && *(pph+1) == 'h')
            {
                break;
            }
        }

        /*
         * no %h, just use the bare client scriptname
         */

        if (*pph == '\0')
```

```
958  2      return temp_scriptname;
959  2
960  2  }
961
962      /*
963       *  there is a 8th ... insert the destination client
964       *  name into the rsc_client_scriptnames buffer.  First
965       *  compute how much storage will be needed to
966       *  hold the result.
967       */
968
969      if (!tbc_can_it_be_normalized(temp_client_hostname))
970  2   {
971  2      rsc_api_log_cmn(SUB_CSM_NOMEM, NULL);
972  2      the_log_state(
973                 0, "Could not allocate memory generate_rcmdpath");
974          StartDoneCallBack(EP_RS_RECOVER_FATALERR);/*
975                 NULL;
976  2      return NULL;
977  2  }
978
979      norm_host = tbc_normalize(temp_client_hostname);
980      len_needed =
981          strlen(temp_scriptname) - 2 +
982          strlen(norm_host) + 1;                  /* -2: %h */
983
984      if (mybuf == NULL || strlen(mybuf) < (size_t)(len_needed-1))
985  2   {                                           /* -2: '\0' */
986  2      free(mybuf);
987
988  2      if (mybuf != NULL)
989  3      {
990  3          mybuf = malloc((int)len_needed)) == NULL)
991  3          {
992  3              rsc_api_log_cmn(SUB_CSM_NOMEM, NULL);
993  3              the_log_state(
994                     0, "Could not allocate memory generate_rcmdpath");
995  3              return NULL;
996  3          }
997  2      }
998
999      q = mybuf;
1000 2   for (q = mybuf, p = temp_scriptname; p < pph; q++, p++)
1001 2   {
1002 3      *q = *p;
1003 2   }
1004 1   (void)strcpy(q, norm_host);
1005 1   (void)strcat(mybuf, pph+2);
1006 1   return mybuf;
1007 1 }   /* end of generate_rcmdpath() */
```

```
1009 1 int
       StartWorkItemRestore(struct recover_context **rcx,
                            struct submit<obj> *p,
                            int SubmitElemID,
                            int SubmitElemID)
       {
           char    *rcmd[4];  /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */
           int     temp_effective_priv = 0;
           int     xcpio_pgm_pid;
           rcmd_pkt0_info pkt0_buffer;
           boolean_ty temp_src_sysadmin;
           char    *err_str_buffer = NULL;

           int GetSESStatus = 0;
           int GetSOStatus = 0;

           char temp_client_hostname[SUBMIT_FIELD_MAX];
           char temp_client_rcname[SUBMIT_FIELD_MAX];
           long temp_effective_uid;
           boolean_ty temp_src_sysadmin;

1024 1     if ( GetSIDestClientName(SubmitToObjectID, SubmitElemID,
1025 2             temp_client_hostname,
1026 2             SUBMIT_FIELD_MAX,
1027 2             &GetSESStatus) != 0)
1028 1     {
1029 2         rbc_log_state(0, "Unable to get client destination name.");
1030 2         return -1;
1031 1     }
1032
1033
1034 1     if ( GetSEClientUserName(SubmitToObjectID, SubmitElemID,
1035 2             temp_client_rcname,
1036 2             SUBMIT_FIELD_MAX,
1037 2             &GetSESStatus) != 0)
1038 1     {
1039 2         rbc_log_state(0, "Unable to get client user name.");
1040 2         return -1;
1041 1     }
1042
1043
1044 1     temp_src_sysadmin = GetSOSourceSystemAdmin(SubmitToObjectID,
1045 2             &GetSOStatus);
1046 1     if (0 != GetSOStatus)
1047 1     {
1048 2         rbc_log_state(0,
1049                 "Unable to get the source system admin privileges.");
1050 2         return -1;
1051 1     }
1052
1053 1     if ( GetSOEffectiveUID(SubmitToObjectID)
1054 2             &temp_effective_uid,
1055 2             &GetSOStatus);
1056 1     if (0 != GetSOStatus)
1057 1     {
1058 2         rbc_log_state(0, "Unable to get effective uid.");
1059 2         return -1;
1060 1     }
1061
1062 1     if ( (0 != (temp_src_sysadmin)) || 0 != temp_effective_uid)
1063 2         set_recovery_privileges(rcx, &changed_priv);
```

```
1074 1          rcmdv[0] = temp_client.hostname;
1075 1          rcmdv[1] = temp_client.rbuname;
1076 1          rcmdv[2] = temp_client.rbuname;
1077 1          rcmdv[3] = make_remote_cpiogen_cmd(rcx,
1078 1                                        SubmitObjectID,
1079 1                                        SubmitElemID);
1080
1081 1      if(NULL == rcmdv[3])
1082 2          return -1;
1083 1      }
1084 1
1085 1      xcpiogen_pid = start_cpiogen(rcx, Xp,
1086 1                              SubmitObjectID,
1087 1                              SubmitElemID,
1088 1                              0,
1089 1                              rcmdv,
1090 1                              &pktdp_buffer,
1091 1                              &err_get_buffer);
1092
1093 1      if (changed_priv)
1094 2          reset_recovery_privileges(rcx, changed_priv);
1095
1096 1      return xcpiogen_pid;
1097 1  }
1098 1  /* StartWorkItemRestore() */
```

```
1100        /*
1101         * Check to see if the user has root access to the
1102         * destination client. If so, give him/her the root
1103         * privileges on the recovery.
1104         */
1105 static void
1106 set_recovery_privileges(struct recover_context *rcx,
1107                          int *changed)
1108 {
1109 1      int root_access;
1110 1      eperrno errnum;
1111
1112 1      *changed = 0;
1113 1      (void)rbc_canirecover(rcx->rc_config, rcx->rc_client.hostname,
1114 1                            rcx->rc_bunm1.uidname, &root_access,
1115 1                            &errnum);
1116
1117 1      if (root_access)
1118 2      {
1119 3          if (debugmode)
1120 4          {
1121 4              rbe_log_stat(
1122 5                  0, "the user is a sys admin for the dest client");
```

```
1122 3          }
1123
1124 2          rcx->rc_recovery_flags |= RC_RECFLAG_DEST_SYSADMIN;
1125 2          if (rcx->rc_effective_uid != 0)
1126 3          {
1127 4              if (debugmode)
1128 5              {
1129 4                  rbe_log_stat(0, "changing the uid to admin status");
1130
1131 3              }
1132 3              rcx->rc_effective_uid = 0;
1133 3              rcx->rc_effective_uidname = "root";
1134 3              *changed = SET_ROOT;
1135 2          }
1136 1      }
1137 1      else
1138 2      {
1139 3          if (debugmode)
1140 4          {
1141 4              rbe_log_stat(
1142 5                  0, "effect uid = %d", rcx->rc_effective_uid);
1143 3          }
1144
1145 2          /*
1146 2           * The user does not have root access to the dest
1147 2           * client. But s/he is the system admin for the
1148 2           * source client, therefore, we need to strip the
1149 2           * root stuff from the user during the recovery.
1150 2           */
1151 2          rcx->rc_recovery_flags &= ~RC_RECFLAG_DEST_SYSADMIN;
1152 2          if (rcx->rc_recovery_flags & RC_RECFLAG_SOURCE_SYSADMIN)
1153 2          {
1154 2              if (debugmode)
1155 4              {
1156 4                  rbe_log_stat(
1157 4                      0, "changing the uid to regular user status");
1158 3              }
```

```
1160 3          rcx->rc_effective_uid = rcx->rc_human_uid;
1161 3          rcx->rc_effective_uidname = rcx->rc_human_uidname;
1162 3          *changed = SET_USR;
1163 2      }
1164 1   }
1165     /* end of set_recovery_privileges() */
```

```
1166     /*
1167      * Reset the user's identity.  The user may have root access
1168      * on the destination client, but not on the source, and vice
1169      * versa
1170      */
1171     static void
1172     reset_recovery_privileges(struct recovery_context *rcx,
1173                               int changed)
1174     {
1175 1       if (changed == SET_ROOT)
1176 1       {
1178 1           if (debugmode)
1179 2           {
1180 2               rbe_log_stats(
1181 3                   0, "resetting the uid to regular user status");
1182 3           }
1183 2
1185 2           rcx->rc_effective_uid = rcx->rc_human_uid;
1186 2           rcx->rc_effective_uidname = rcx->rc_human_uidname;
1187 1       }
1188 1       else
1189 1       {
1190 2           if (debugmode)
1191 2           {
1192 3               rbe_log_stats(0, "resetting the uid to sys admin status");
1193 2           }
1195 2           rcx->rc_effective_uid = 0;
1196 2           rcx->rc_effective_uidname = "root";
1197 1       }
1198     }
         /* end of reset_recovery_privileges() */
```

```
1202    /**
1203    **
1204    ** FUNCTION DESCRIPTION:
1205    **
1206    **   This function will start the workitem restore termination.
1207    **
1208    **
1209    ** INPUTS:
1210    **   auxproc_pid -- auxproc's pid.
1211    **
1212    **
1213    ** RETURN VALUE:
1214    **   none
1215    **
1216    ** SIDE EFFECTS:
1217    **
1218    **   auxproc sent the USR1 signal, auxproc will send xcpiogen the
1219    **   USR1 signal to quit the restore. Auxproc will wait until the
1220    **   restore signal is sent to auxproc. auxproc will notify the
1221    **   status terminates by waiting for the remote command's exit
1222    **   status
1223    **
1224    **
1225    **/
1226
1227    void
1228    QuitWorkItemRestore(int auxproc_pid)
1229    {

1232        /*
1233         * Alert the auxproc that we went out.
1234         * xcpiogen etc should also die as a result
1235         * of xx_read_or_die_xx() in recv. The SIGUSR1
1236         * should not kill the auxproc itself, but only
1237         * xcpiogen will commit suicide as a result of
1238         * this signal.
1239         */

1241        /*
1242         * This will give xcpiogen the ability to
1243         * clean up tmpfiles and clean-up sockets.
1244         * xcpiogen will commit suicide as a result of
1245         * notify auxproc of the diminishing restore.
1246         * this signal.
1247         */

1249        (void)kill(auxproc_pid, SIGUSR1);

1251        if (debugmode)
1252        {
1253            the_log_stats(0, "%s %d quitting restore in process",
1254                    AUXPROCNAME,
1255                    auxproc_pid);
1256        }

1258        /*
1259         * now that we have indirectly killed the xcpiogen and alerted the
1260         * auxproc,
1261         * the signal from the auxproc will be picked up by the next
1280         * routine. auxprocsig_handler, it will notify the user of the
1281         * results and does the cleaning up.
1282         */
```

```
1263    return;
1264    } /* QuitWorkItemRestore() */
1265
```

```
1267  /**
1268   **
1269   **  GetAuxprocResults()
1270   **
1271   **  Inherited from auxproc:rslg_handler()/.
1272   **
1273   **  This routine is called when there is information appears
1274   **  in the aux-process. The aux-process starts xplogen and
1275   **  is responsible for listening to status coming from xplogen.
1276   **  When status comes back from xplogen, the aux-process signals
1277   **  this process, which is trapped by the caller. And finally,
1278   **  this routine is called.
1279   **
1280   **  Args:
1281   **      (i) rsetd int -- results file descriptor from auxproc.
1282   **      (o) results -- work item results.
1283   **
1284   **  RETURN VALUE:
1285   **      The number of local and remote status collected from fd.
1286   **
1287   **  SIDE EFFECTS:
1288   **      none
1289   **
1290   **
1291   **
1292   **/

1295  int
1296  GetAuxprocResults(int rsetd,
1297                    wi_restore_results *results)
1298  {
1299      int      exit_stat;
1300      char     *rebuf;
1301      int      n_read;
1302      char     c = 0;
1303      int      n_status_read = 0;
1304
1305      while ((n_status_read < 2) && (1 == fd_avail_test(rsetd)))
1306      {
1307          rebuf = (char *) &exit_stat;
1308
1309          if(1 != pread_or_warn(rsetd, &c, 1, auxproc_comm_warning))
1310          {
1311              return 0;
1312          }
1313
1314          if (debugmode)
1315              the_log_stats(0, "GetAuxprocResults() called 'tc'", c);
1316
1317          if (c == 'R')
1318          {
1319              /*
1320               * the 'R' command is the for the remote process status
1321               */
1322              n_read = auxres2(rsetd, 'R', sizeof(int), &rebuf);
1323              if (-1 == n_read )
```

```
1325              {
1326                  if (debugmode)
1327                      the_log_stats(
1328                          0, "remote exit status obtained: %d", exit_stat);
1329              }
1330              else
1331              {
1332                  the_log_stats(
1333                      0, "Internal error: remote exit status incomplete.");
1334                  return -1;
1335              }
1336              results -> remote_exit_status = exit_stat;
1337              results -> remote_exit_set = TRUE;
1338              n_status_read++;
1339          }
1340          else if (c == 'r')
1341          {
1342              /*
1343               * the 'r' command is the for the local process status
1344               */
1345              n_read = auxres2(rsetd, 'r', sizeof(int), &rebuf);
1346              if (-1 != n_read )
1347              {
1348                  if (debugmode)
1349                      the_log_stats(
1350                          0, "local exit status obtained: %d", exit_stat);
1351              }
1352              else
1353              {
1354                  the_log_stats(
1355                      0, "Internal error: local exit status incomplete.");
1356                  return -1;
1357              }
1358              results -> local_exit_status = exit_stat;
1359              results -> local_exit_set = TRUE;
1360              n_status_read++;
1361          }
1362      }
1363      /* while() */
1364
1365      /*
1366       * sleep (1);
1367       */
1368
1369      return n_status_read;
1370      /*return n_status_read;*/
1371  }
1376  /* GetAuxprocResults() */
```

```
1379    /*
1380     * KillWorkItemRestore()
1381     *
1382     * Kill the work item restore. Keep in mind this
1383     * may only be done if the work item is not running
1384     * a restore.
1385     *
1386     * This routine also does the waitpid for auxproc.
1387     * The waitpid (ChildDone()) eliminates the defunct auxproc
1388     * process.
1389     *
1390     * If the work item is running then one must do the
1391     * following:
1392     *
1393     *   1) Call QuitWorkItemRestore()
1394     *   2) Wait for and read results from the cmd_from pipe.
1395     *   3) Call KillWorkItemRestore()
1396     *
1397     * Args:
1398     *   ap_pid   -- auxproc pid.
1399     *   cmd_to   -- auxproc cmd from pipe.
1400     *
1401     * Returns:
1402     *   int  -- zero for success.
1403     */
1404    int
1405    KillWorkItemRestore(int ap_pid, int cmd_to)
1406  1 {
1407  1     int killRet;
1408  1     int apResult;
1409  1     char *apName=AUXPROCNAME;
1410  1     char *debuf = NULL;
1411  1     int ChildDoneRet;
1413  1     killRet = kill(ap_pid, SIGTERM);
1415  1     if (-1 == killRet)
1416  1     {
1417  2         the_log_stats(
1418  2             0, "Can't send sigterm to \"%s\", pid: %d, error = %s\n",
1419  2             apName, ap_pid, strerror(errno));
1421  2         return -1;
1422  1     }
1424  1     do
1425  1     {
1426  2         ChildDoneRet = ChildDone(ap_pid, &apResult);
1427  2         if(0 == ChildDoneRet) sleep (1);
1429  1     } while(0 == ChildDoneRet);
1431  1     switch (ChildDoneRet)
1432  1     {
1433  2         /*  -1 internal error. errno is set.
1434  2          *   0 child still running
1435  2          *   1 child exited.
1436  2          *   2 child signaled (no core)
1437  2          *   3 child signaled core file generated.
1438  2          *   4 child signaled core stopped.
1439  2          */
1440  2         case(-1):
1441  2             return -1;
1442  2             /* no break necessary */
```

```
1443  2         case(1):
1444  2             the_log_stats(0,
1445  3                 "Sigterm did not bring down \"%s\", pid: %d."
1446  3                 " It instead exited with = %d\n",
1447  3                 apName, ap_pid, apResult);
1448  2             break;
1451  2         case(2):
1452  2         case(3):
1453  3             if (SIGTERM != apResult)
1454  3             {
1455  3                 the_log_stats(0,
1456  3                     "Sigterm did not bring down \"%s\", pid: %d"
1457  3                     " instead killed by signal = %s\n",
1458  3                     apName, ap_pid,
1459  3                     strsignal(apResult));
1460  3             }
1461  2             break;
1463  2         case(4):
1464  3             the_log_stats(0,
1465  3                 "Sigterm did not bring down \"%s\", pid: %d"
1466  3                 " instead stopped by signal = %s\n",
1467  3                 apName, ap_pid,
1468  3                 strsignal(apResult));
1470  2             break;
1472  2     }
1475  2     /* auxcmdpacket(cmd_to, "q", 0, databuf);*/
1476  2     return 0;
1478  2 }  /* KillWorkItemRestore() */
```

```c
 1
 2   *
 3   *   Copyright (c) 1998, 1999 EMC Corporation.  All rights reserved.
 4   *
 5
 6   *   Table of Contents:
 7   *
 8   *   static char *rbl_getmethod(host, int *concurrency, int sigvalue);
 9   *   static void write_buf, int nbytes);
10   *   int logvalue[] = { wait_interim(int fd);
11   *   int fd_avail[] wait_interim(int fd);
12
13
14   #define __POSIX_SOURCE 1
15   #define E_GRANDFATHER 1
16   #define TIME_STRUCT
17   #include <eb/ep_port.h>
18
19   #include <netdb.h>
20   #include <pwd.h>
21   #include <ctype.h>
22   #include <sys/wait.h>
23   #include <sys/select.h>
24   #include <errno.h>
25   #include <util/sel_strerror.h>
26   #include <util/sel_limit.h>
27   #include <stdlib.h>
28
29   /* SFP header */
30   #include <cdl/cdl.h>
31   #include <cdl/cdl_server.h>
32
33   #include <bbconfig/rbconfig.h>
34   #include <epfdhb/ffdb.h>
35   #include <epfdb/rbfinclient.h>
36   #include <util/eb_nonalloc.h>
37   #include <ebull/ebuild.h>
38   #include <ebull/ebuild.h>
39   #include <eb/rb_log.h>
40   #include <restore/RSprograms.h>
41
42   #include <RSLintercom.h>
43   #include <RStapexits.h>
44   #include <RSEucomp.h>
45   #include <RSLauxhop.h>
46   #include <edm/link_api.h>
47
48   /* EDMLINK API */
49
50   /* line ugliness */
51   #ifndef Printf
52   #define Printf  (void)printf
53   #endif
54   #define MAX_FD 1024
55   int debugmode = 0;
56   int sb_upgmask = 0;
57   int sp_collectdone = 0;
58
59
60   static int xpplogen_pid = -1;
61   static int xpplogen_prog_id = -1;
62   int log/msg_channel = -1;
63   int xpplogen_pipe[2] = {-1,-1};
64   static char wifiedit[] = "/usr/epoch/etc/";
65
66   static char *xpplogen_pipe(
```

```c
67   register char *host, uint_t *concurrency, uint_t *client_type)
     static void sigterm_handler(int sigvalue);  /* extension signal */
     static void sigterm_handler(int sigvalue);

     static void write_CDL_no_eint(int fd, char *buf, int nbytes);
     static int read_CDL_io_eint(int fd, char *buf, int nbytes);

     static int ForwardXpplogenProgress(int xpplogen_prog_id,
                                        int xpplogen_prog_fd,
                                        boolean_t restore_engine_prog_fd,
                                        boolean_t *zero_byte_read);

     static int DemuxAuxChildren(int progress_fd,
                                 int remote_progname,
                                 int xpplogen_fd,
                                 int xpplogen_pid,
                                 int *remote_exit);

     extern int fd_avail_test(int);  /* Found in RSLauxSupp.c */

     struct auxproc_context {
         int         ap_my_auxnum;
         int         ap_??_fd;
         char        *remote_progname;
         int         ap_v_fd;
         int         ap_shellrcp_port;
         int         ap_w_prog_fd;
         int         ap_datalen;
         char        ap_data[];
         char        ap_cmd;
         int         ap_datalen;
         int         ap_resultlen;
         char        *ap_results;
         char        *ap_results1data;
         ushort_t    ap_shellrcp_port;
         int         ap_havshellrcp_port;
         int         ap_socketport;
         char        *ap_sockethost;
         int         ap_v_fd;
         char        *ap_workitem;
         struct      rbc_config *ap_config;
         char        ap_error_message[4096];
     };

     /*
      * Read remote stderr output from fd.
      * Parse stderr etc. etc according to protocol.
      * Remote exit status is the last thing that comes
      * back via the remote stderr stream; return the remote
      * exit status via *exitcp.
      */
```

```c
     enum input_states

         INSTATE_MSG,
         INSTATE_SEARCH_PREFIX0,
         INSTATE_SEARCH_PREFIX1,
         INSTATE_SEARCH_PREFIX2,
         INSTATE_GATHER_COOKIE,
         INSTATE_SEARCH_SUFFIX0,
         INSTATE_GATHER_STATUS,
         INSTATE_COPY_TO_STDOUT,
         INSTATE_NEWLINE
     };

     static boolean_t parse_remote_stderr_info2(int prog_fd,
                                                int remote_fd,
                                                char *remote_name,
                                                boolean_t *embedchunks,
                                                boolean_t *first_write_call,
                                                enum input_states
                                                *state_ptr,
```

```
                    enum input_states
                        boolean_ty       next_state_ptr,
                    *skipping_leading_whitespace,
                        int *parsebos,
                        int *msgPos);

130
131
132
133

136     static char  *recover_size_prefix(struct auxproc_context *cxp);
137     static int    *ebr_direct_rcmd(char *abdot, unsigned short import,
138                     struct auxproc_context *cxp,
139                         char *locxuser,
                            char *remuser, char *cmd, int *fdzp);
```

```
141     /* The auxiliary process(es) communicate with the main
142      * process via a simple protocol run on a pair of pipes.
143      *
144      * The parent writes commands to the auxiliary process.
145      * The format of a command is:
146      *
147      *      <cmd><data-len><data>
148      *
149      * where
150      *      <cmd>    is a one-byte command
151      *      <data-len> is an 'int', and indicates the number of <data> bytes
152      *
153      *      <data>   is command-specific data
154      *
155      * <data-len> may be zero, but must always be present. Therefore, the
156      * minimum command 'packet' is five bytes long.
157      *
158      * Result packets are written back to the parent.  The packet format is
159      * the same as the command format, though (obviously) the format
160      * of the data in a result packet is usually different from
161      * the format in a command packet.
162      *
163      * Communications are assumed to be error-free.  In other
164      * words, pipes are assumed to work correctly.
165      */
167     static int attn_ec;        /* count of SIGUSR1s (ATTN signals) */

171     /*
172      * 'z' prefixes identify top-level routines
173      * called from the auxproc switch.
174      *
175      * There is no significant to the 'z' letter -- I just picked
176      * a letter at random (hah!) to identify the top level func.
177      */
179     static void z_rcmdfilter(struct auxproc_context *cxp);
180     static void z_exec_separator(struct auxproc_context *cxp,
181                     char *pathname);
```

```
183   int main(int argc, char *argv[])
184 1 {
185 1     char *auxproc_intial_delay_str = NULL;
186 1     int auxproc_intial_delay_int = 0;
187
188 1     if (argc != 9)
189 1     {
190 2         exit(1);
191 1     }
192
193 1     auxproc_intial_delay_str = getenv("AUXPROC_INITIAL_DELAY");
194 1     if (NULL != auxproc_intial_delay_str)
195 1         auxproc_intial_delay_int = atoi(auxproc_intial_delay_str);
196 1         sleep(auxproc_intial_delay_int);
197 1     }
198
199
200 1     debugmode = atoi(argv[6]);
201
202 1     do_auxproc(atoi(argv[1]),     /* auxproc ordinate */
203 1         atoi(argv[2]),     /* cmd_to auxproc pipe*/
204 1         atoi(argv[3]),     /* cmd_from auxproc pipe*/
205 1         atoi(argv[4]),     /* bulk_to auxproc pipe*/
206 1         atoi(argv[5]),     /* prog_from auxproc pipe*/
207 1         argv[6],           /* Mname */
208 1         argv[0],           /* programe */
209 1         argv[7],           /* socket host name */
210 1         atoi(argv[8]));    /* socket port */
211
212
213       return(0);   /* Can we get here ?? */
      } /* End main() */
```

```
215   /*
216    * Worker-bee loop.
217    * Read instructions on r_fd.
218    * Write results to w_fd.
219    */
220
221
222   int ncmds_rcvd = 0;
223
224 1 void
      do_auxproc(int procnum,
225             ...
226             int w_fd,
227             int r_bulk_fd,
228             int w_prog_fd,
229             char *Mname,
230             char *socKethost,
231             int socketport)
232   {
233 1     struct auxproc_context    ctx;
234 1     struct current            *spi;
235       sigset_t                  empty_set;
236       sigset_t                  ...
237 1     int                       rotation;
238 1     int                       errnum;
239 1     int                       index_fd;
240
241 1     /*
242 1      * close all file descriptors that we do not need
243 1      * we only need the ones passed to us from the
244 1      * engine.  So we can close anything above stderr and anything
245 1      * that is no equal to what was passed in
246 1      */
247 1     for(index_fd = 3; MAX_FD > index_fd; index_fd++)
248 1     {
249
250 2         if ((index_fd == r_fd) ||
251 2             (index_fd == w_fd) ||
252 2             (index_fd == r_bulk_fd) ||
253 2             (index_fd == w_prog_fd))
254 3         {
255 3             continue;
256 2         }
257 2         (void)close(index_fd);
258       }
259 1     sigemptyset(&empty_set);
260
261 1     memset(&ctx, 0, sizeof(struct auxproc_context));
262 1
263 1     /*
264 1      * Prepare sigaction parameters
265        */
266 1
267       /*
268 1      * Ignore mild keyboard interrupts ("C); parent handles
269        */
270 1
271       /*
272 1      * SIGUSR1 used to get our attention when the parent
273        * processes really wants us to stop what we are doing.
274 1      */
275 1     eb_set_signal_handler(SIGINT, SIG_IGN, &empty_set, E_SA_RESTART);
276 1
277 1
278
279 1     eb_set_signal_handler(
```

```
280      eb_set_signal_handler( SIGUSR1, signal_handler, &empty_set, E_SA_RESTART);
                                 SIGTERM, sigterm_handler, &empty_set, E_SA_RESTART);

282      /*
283       * Arguments passed are collected together in
284       * this context structure only because that makes
285       * it easier to add/change arguments in the future
286       * (diddle the structure rather than diddle a zillion
287       * calls in the switch statement).
288       */

290      ctx_ap.my_auxnum  = procnum;
291      ctx_ap.r_fd       = r_fd;
292      ctx_ap.w_fd       = w_fd;
293      ctx_ap.r_bulk_fd  = r_bulk_fd;
294      ctx_ap.w_prog_fd  = w_prog_fd;
295      ctx_ap.sockethost = sockethost;
296      ctx_ap.socketport = socketport;

298      /*
299       * Will need the shell/tcp service later; if
300       * can't find it now we might as well compile now.
301       */
303      ctx_ap.have_shelltcp_port = 0;
305    1 if ((sp = getservbyname("shell", "tcp")) != NULL)
306    1 {
307    1     ctx_ap.shelltcp_port = (ushort_t)sp->s_port;
308    1     ctx_ap.have_shelltcp_port = 1;
309    1 }
310    1 else
312    2 {
313    2     WritePmtStringMsg(
314    2         ctx_ap.w_prog_fd, EDMKERRORMSG_AUXPROC_ERROR, 0,
315    2         "Error: Cannot find shell/tcp network
316    2         service.\n"
318    2         "Browsing of catalogs may continue, but
319    2         the \"start\" command will not work.");
320    2 }

321  1 if (NULL == ctx_ap.config)
323  2 {
324  2     /*
325  2      * We would prefer to log this stuff, but we
326  2      * have not opened logging yet.
327  2      */
328  2     exit(1);
330  2 }

332  2     if (rbc_parse_config(
333  3         NULL, /*use the default name */, &ctx_ap.config,
334  3         RBC_PARSE_DO_NOT_PRESERVE |
335  3         RBC_PARSE_APPLY) != 0)
```

```
337  3         rec_apl_log_cmn(SUB_CSM_NO, PARSE_CFG, NULL);
338  3         WritePmtStringMsg(
339  3             ctx_ap.w_prog_fd, EDMKERRORMSG_AUXPROC_ERROR, 0,
                     "Auxproc -- Cannot parse config
                     file");
340  3         exit(1);
342  2     }

344  2     if ((errnum = eb_path_init()) != 0)
345  2     {
346  2         exit(1);
347  2     }

348  1     (void)the_log_init("auxproc");

351  1     /*
352  1      * Determine the "recovery.log" file rotation size.
353  1      * Use a default unless this was specified in the config file.
354  1      */
356  1     rotation_size = RBRECOVER_LOGSIZE;
357  1     if (ctx_ap.config != (struct rbc_config *)NULL)
358  1         && (ctx_ap.config->rsfile->filename != NULL)
359  1         rotation_size = ctx_ap.config->rsfile->rotation.rotation_size;

361  1     (void)the_log_add(STATE_LOGGING, eb_recover_logpath, LOGT_FILE,
364  1         rotation_size, &logging_channel);

368  1     /*
369  1      * The actual worker-bee loop
                */

371  2     for (;;)
372  2     {
373  2         char c;
374  2         int datalen;
             #define SMALL_DATALEN 128
375  2         char buf[SMALL_DATALEN];
376  2         char *data;

377  2         /*
379  2          * read command byte
380  2          */
381  2         pread_or_die(r_fd, &c, 1, _exit);

383  2         pread_or_die(r_fd, &c, 1, _exit);

387  2         pread_or_die(r_fd, (char *)&datalen, sizeof datalen, _exit);

389  2         /*
391  2          * We do the read here for single commands that do not
392  2          * take much data.
393  2          */
395  2         if (datalen > SMALL_DATALEN || datalen == 0)
396  2             data = NULL;
397  2         else
398  2         {
400  2             data = ...
```

```c
401        {
402  2         pread_or_die(r_fd, buf, datalen, _exit);
403  2         data = buf;
404  2       }

406  2       ++cmds_rcvd;
407  2       ctx.ap_cmd        = c;
408  2       ctx.ap_datalen    = datalen;
409  2       ctx.ap_data       = data;
410  2       ctx.ap_resultlen  = 0;
411  2       ctx.ap_resultdata = NULL;

413  2       /* NOTE: This switch statement is in alpha-order
414  2        *       (case-folded) to make it easier for humans
415  2        *       to find entries (at least, it was in alpha-order
416  2        *       when I wrote this comment).
417  2        */
418  2
             switch (c)
420  2       {
421  2         /* 'P' */
422  2
423  2         /* 'p' */
425  3           /*
                  *  Ping command.  Used for debugging.
                  */

427  3           ctx.ap_resultlen  = ctx.ap_datalen;
429  3           ctx.ap_resultdata = ctx.ap_data;
430  3           break;

         case 'q':
432  3           /* 'q' */
433  3           /*
434  3            *  Quit command.
                  */

436  3           rbe_close_logs(logging_channel);
437  3           _exit(0);
438  3

         case 'r':
440  3           /*
441  3            *  'r'
442  3            *
443  3            *  rand filter.  Fire up a process, typically xcgslogen,
444  3            *  with output from the process going to an rcmd
445  3            *  connection.
                  */

         case 'r':
447  3           z_protillbar(&ctx);
448  3           break;
449  3
                 /*
451  3            *  'x'
452  3            *
453  3            *  Exec separate process version of do_auxproc.
454  3            *  Used for debugging.
455  3            *  No value returned to parent.
456  3            *  If exec fails, auxproc dies.
                  */

         case 'X':
458  3           z_exec_separate_auxproc(&ctx, Xname);
459  3           break;
460  3
         default:
461  3           rbe_apl_log_cmn(SUB_CBM_INV_AUXPROC_CMD, NULL);
463  3           rbe_log_state( 0,   "auxproc -- invalid command in do_auxproc");
464  3           exit(2);
465  3           break;
```

```c
466  3           exit(1);
467  3           break;
468  2       }    /* end of switch */

470  2       /*
471  2        *  Simple commands set ap_resultlen and ap_resultdata
472  2        *  and we push the results to the parent here.
473  2        */

475  2       if (ctx.ap_resultlen > 0)
476  2       {
477  2          pwrite_or_die(w_fd, &c, 1, _exit);
478  2          pwrite_or_die(w_fd, (char *)&ctx.ap_resultlen,
479  2                        sizeof ctx.ap_resultlen, _exit);
480  2          pwrite_or_die(w_fd, ctx.ap_resultdata,
481  2                        ctx.ap_resultlen, _exit);
482  2       }

484  2       if('r' == c)
485  2          rbe_log_stats(0, "auxproc(%d) local exit is %d.", getpid(),
486  2                        ctx.ap_resultlen);
487  2       }

#if 0
489  2       }
490  2 #endif
491  1     }    /* end of for loop */
492       }      /* end of do_auxproc() */
```

```
494  /*
495   * Fire up a process with stdin from parent and output to rcmd.
496   * Protocol traffic between 'parent' (main process) and us:
497   *
498   *       'r' parent.  --> auxproc      contains local & remote info
499   *       'q' auxproc  --> parent       returns 'setup' result, see below
500   *       'R' auxproc  --> parent       returns remote exit status (1 int)
501   *       'r' auxproc  --> parent       returns local exit status (1 int)
502   *
503   * The '0' result packet describes the results of setting things up.
504   * It contains the following information:
505   *
506   *       success/failure : int   [ 0 is success,
507   *                                 non-zero is a (failure code) ]
508   *       errnum          : int   [ 0 if no applicable errno code ]
509   *       pid             : int   [ process ID of local xloproc proc ]
510   *
511   *       msglen          : int   [ length, in bytes,
512   *                                 of following str ]
513   *       errstr          : string [ string describing failure ]
514   *
515   * If things were set up successfully, then two integer zero
516   * values, a non-zero pid, and one more zero (msglen) are
517   * sent in the '0' packet.
518   *
519   * The 'R' result packet will not be sent if the '0' and 'r' result
520   * indicates that an error occurred. The '0' and 'r' result
521   * packets are always sent.
522   *
523   * parent passes the following in the 'r' packet 'data':
524   *
525   *       (string) rcmd-hostname
526   *       (string) rcmd-rcmdee
527   *       (string) rcmd-cmd
528   *       (int)    setuid-val
529   *       (int)    filepac from workitem
530   *       (int)    filter-cmd-fd-info [explained below]
531   *       (int)    future-flags [flags for future hacks]
532   *                                 always zero now]
533   *
534   *       (string) filter-cmd
535   *       (int)    filter-cmd-argc
536   *       (string) filter-cmd-argv0
537   *       (string) filter-cmd-argvN
538   *       (int)    rcmd-cmd
539   *       (string) rcmd-renamer
540   *       (string) rcmd-hostname
541   *       (int)    db API socket.
542   *       (string) db socket. host name
543   *       (int)    filepac socket.
544   *
545   * If filter-cmd-fd-info is -1, then filter-cmd-argv1
546   * should be a sprintf format string which will be given to
547   * sprintf along with an integer to produce the
548   * file descriptor number to pass the filter command.
549   *
550   * Items which are (string) are '0' terminated.
551   *
552   * Parent is responsible for obeying the built-in size limits:
553   *
```

```
555   *       no more than 100 argv strings for filter-cmd
556   *       no more than 10000 total bytes of auxproc data
557   *       no more than 100 bytes of filter-cmd-fd-info argv
558   */
559
560  struct rcmd_pkt0_info {
561      int failcode;
562      int errnum;
563      int pid;
564      int msglen;
565      /* variable length char string message follows */
566  };
567
568  /*
569   * Function to build command line prefix for adding environment variable
570   */
571
572  /* for log truncation */
573
574  static char lbuff[128];
575
576  static char *
577  recover_size_prefix(struct auxproc_context *cxp)
578  {
579      if (cxp->ap_config->crtfile_rotation_size != NO_ROTATION)
580          /* export EB_MAX_CLIENT_LOG_SIZE=%d */
581          sprintf(lbuff, "EB_MAX_CLIENT_LOG_SIZE=%d,",
582                  cxp->ap_config->crtfile_rotation_size);
583      else
584          memset(lbuff, 0, sizeof(lbuff));
585
586      return lbuff;   /* recover_size_prefix */
587
588  }
```

```
static void
z_rcmdfilter(struct auxproc_context *cxp)
{
    char    *data;
    char    rcmd_stuff[4];
    char    (                       /* 0 hostname, 1 locuser, 2 remuser, 3 cmd */

    char    *human_name;
    struct  rcmd_pkt0 pkt0;
    static  char r_resultstr;
    static  int  r_resultcd;
    int     filter_cmd_argc;
    char    *filter_cmd_argv;
    int     filter_cmd_fd_info;
    int     filter_cmd_fdout;
    char    *filter_cmd;
    char    *method;
    int     dbsegno=0;
    int     socket_port;
    int     socket_host[100];
    char    *socket_file;
    char    wlfilename[200];
    FILE    *infofd;

static char readmode[] = "r";       /* items for socket info file --- */
    char    cbuf[200];
    char    *listener_filename[DB_MAXPATHLEN];
    char    *workitem;
    int     rcmd_scderr = -1;
    int     *rcmd_fd;
    int     rcmd_fds[2];
    int     L,m;
    char    c;
    char    p2;
    int     wait_result;
    char    fdarg[100];             /* BUILD-IN LIMIT */
    char    *rcmd_argv[100];        /* BUILD-IN LIMIT */
    char    databuf[100000];        /* BUILD-IN LIMIT */
    char    *filespec;
    char    selbuf[200];            /* skip the flags */

    char    holdbuf[1024];          /* for command with ssl parameter */
                                    /* hold thrclient ... command */

    RBC_WORKITEM *pwi;
    RBC_WORKGROUP *pwg;
    boolean_ty xcplogon_still_running;
    e_errno_ty errrnum;

    /* EDMLINK handle */
    ELinkHandlert_ty        pELinkHandle    = NULL;

    /* EDMLINK objects */
    ELinkTargetObjsRet_ty   pELinkTargetObj = NULL;
    ELinkUserIdObjsRet_ty   pELinkUserIdObj = NULL;
    ELinkCmdObjsRet_ty      pELinkCmdObj    = NULL;

    /* EDMLINK global options */
    unsigned long ELinkOptions = 0;

    /* EDMLINK status */
    int ELinkStatus = 0;
    int ELinkStatus_errno = 0;

    data = cxp->ap_data;
```

```
    if (data == NULL)
    {
        /*
         * command dispatcher did not read data for us,
         * there was too much. We must read it.
         */

        data = databuf;
        pread_or_die(cxp->ap_r_fd, data, cxp->ap_datalen, _exit);
    }

    /*
     * extract the 4 rcmd strings.
     * [3] is the recmpio cmd.
     */

    for (i = 0; i < 4; i++)
    {
        rcmd_stuff[i] = data;
        data += strlen(rcmd_stuff[i])+1;
    }

    /*
     * extract the remote uidname to be used
     */

    human_name = data;
    data += strlen(human_name) + 1;

    /*
     * extract the filter-cmd-info
     */

    memcpy(&filter_cmd_fd_info, data, sizeof (int));
    data += sizeof (int);

    /*
     * skip the flags
     */

    data += sizeof (int);

    /*
     * extract the filter command
     */

    filter_cmd = data;
    data += strlen(filter_cmd)+1;

    memcpy(&filter_cmd_argc, data, sizeof filter_cmd_argc);
    data += sizeof filter_cmd_argc;

    filter_cmd_argv[filter_cmd_argc] = NULL;

    for (i = 0; i < filter_cmd_argc; i++)
    {
        filter_cmd_argv[i] = data;
        data += strlen(filter_cmd_argv[i])+1;
    }

    /*
     * extract db API socket info
     */

    memcpy((char *)&socket_port, data, sizeof (int));
```

```
720  1        data = sizeof (int);

722  1        /*
723  1         * extract db API socket host name
724  1         */
726  1        sockt_file = data;
727  1        data += strlen(socket_file) + 1;

729  1        /*
730  1         * extract filespec
731  1         */
733  1        filespec = data;
734  1        data += strlen(filespec) + 1;

736  1        /*
737  1         * extract workitem
738  1         */
740  1        workitem = data;
741  1        data += strlen(workitem) + 1;
742  1        data = strlen(workitem) + 1;

744  1        pkt0.failcode = 0;
745  1        pkt0.errnum   = 0;
746  1        pkt0.pid      = 0;
747  1        pkt0.msglen   = -1;
748  1        pkt0.errstr   = "";

750  1        /*
751  1         * locate work item in config info
752  1         */
754  1        if (NULL == cxp->cxp_config)

756  2            if(NULL == (cxp->cxp_config = malloc(sizeof(
757  2                            struct rbc_config))))

759  3                rbc_log_state(
760  3                0, "Could not allocate memory in z_rcmdfilter!");
761  3                exit(1);

762  2            if (rbc_parse_config(
763            NULL, /* use the default name */, &cxp->cxp_config,
                    RBC_PARSE_DO_NOT_PRESERVE |
                    RBC_PARSE_APPLY) != 0)

764  3                rbc_log_state(
765  3                rbc_log_state(
766  3                0, "auxproc -- Cannot parse configuration file");
767  3                exit(1);

769            }

771  2        for (pwg = cxp->cxp_config->cxp_groprouplist; ; pwg = pwg->next)
772  2        {
773  2            if (pwg == (RBC_WORKGROUP *)NULL)

774  3                char cxm_err_msg[256];

776  3                rbc_log_state(
777  3                0, "\n\t\t ** No work item name \"%s\" in configuration file",

779  3                sprintf(cxm_err_msg
```

```
780  3            "Cannot restore. Work item not in cb.cfg (
                    No work item for %s.)",
781  3            rbc_api_log_csm(SUB_CSM_NO_WITEM_CFG, csm_err_map)
782  3            rbc_log_state(0, "Cannot continue without a work item");
784  1            exit(1);

787  2        for (pwi = pwg->pwilist; NULL != pwi; pwi = pwi->next)
789  3            if (0 == strcmp(pwi->name, cxp->cxp_workitem))

791  4                goto gotit;

793  2        }
794  1        goto gotit;

796  1        /*
797  1         * connect by direct or rsh methods
798  1         */
799    gotit:

801  1        if ((method = rb_getmethod(rcmd_stuff[0], NULL, NULL)) == NULL)

803  2            WriteUrmStringMsg(
804            cxp->cxp_w_prog_fd, EOMERRPROGMSG_AUXPROC, WARNING, 0,
805            "Unable to get connection method to host",
806  2            "defaulting to \"rsh\" method",
807  2            rcmd_stuff[0]);
808  2            method = "rsh";

810  1        /*
811  1         * now check for a work item override of the connection method
812  1         */
814  1        if (CXMITN_RSH != pwi->connection_type) /* if other than rsh */
816  2            switch (pwi->connection_type)
817  2            {
818  3            case CXMITN_RSH:
819  3                method = "rsh";
820  3                break;

822  3            case CXMITN_PXMLINK:
823  3                method = "admlink";
824  3                break;

826  3            case CXMITN_DIRECT:
827  3                method = "direct";
828  3                break;

830  3            case CXMITN_SOCKET:
831  3                method = "socket";
832  3                break;

834  3            case CXMITN_NETWARE:
835  3                method = "netware";
836  3                break;

838  3            default:
839  3                method = "????";
840  3                break;
842  2            }
```

```
843  3          if (debugmode)
844  3          {
845  3              the_log_stat=(
846  4                  "using connection method \"%s\" due to work item \"%s\" override",
847  4                  method, pvi->name);
848  3          }

850  2          if (NULL != pvi && pvi->recovery_init) {   /* if a recovery init
                                                              command specified */
851  2              (void)system(pvi->recovery_init);
852  2          }
853  1      }

           /* OSSdev369652 -- Workaround for STPclose() behavioral defect
              If pvi->ssl_groupname is not NULL we are going to be using
              SP for the recovery.
              Shut off the SSL (STP) exit handler.
              Environment variable is used here so the call to
              CDL_noatexit can be avoided, if desired.
           */

858  1      if ( NULL != pvi->ssl_groupname )
859  1      {
860  1          putenv("STP_ENVLEVEL=DEBUG2");
861  1          putenv("STP_EVTLOG_SHARE=OFF");
862  1      }

864  1      if ( NULL != getenv("AIXRPROC_DONOT_PERFORM_CULROATEXIT") )
865  1      {
866  1          CDL_noatexit();
867  1      }
       #endif

       #if 0

869  2      if (0 == strcmp(method, "direct"))
870  2      {
           /*
            * direct connection method here
            */
873  3          (void) putenv("STP_EVTLEVEL=DEBUG2");
874  3          (void) putenv("STP_EVTLOG_SHARE=OFF");
875  2      }

877  2      if (0 == strcmp(method, "direct"))
878  2      {
879  2          is_sympath = 1;
880  2      }

882  2      if (0 == strcmp(method,
883  2      {
           /*
            * the_log_stat=(0,"Invoking cmd with %s, %s, %s, %s\n",
            *     rcmd_stuff[0], rcmd_stuff[1],
            *     rcmd_stuff[2], rcmd_stuff[3]);
            */

           strcpy(cbuf, recover_size_prefix(cxp));

897  2      if (strlen(cbuf) > (size_t)0)
898  2      {
899  3          sprintf(holdbuf, "( %s %s )", cbuf, rcmd_stuff[3]);
900  3          rcmd_stuff[3] = holdbuf;
901  2      }

903  2      rcmd_fd = exr_direct_rcmd( rcmd_stuff[0], cxp->xp_shellcp_port, cxp,
904  2                  &rcmd_stuff[0], rcmd_stuff[1], rcmd_stuff[2], cxp,
                        rcmd_stuff[3]);
```

```
905  2      if (debugmode)
906  3      {
907  3          the_log_stat=(
908  3              0, "AIXRPROC: rcmd returned fd %d, stderr fd %d\n",
909  3              NULL != rcmd_fd[0] ? rcmd_fd[0] : -1, rcmd_stderr[0]);
910  3      }

912  2      if (rcmd_fd == NULL)
913  2      {
914  3          pkt0.failcode = 2;
915  3          pkt0_errstr = "cannot set up remote connection";
916  2          goto send_0;
917  2      }
918  1      else if ( (0 == strcmp(method, "rsh")) || (0 == strcmp(
919  1                  method, "xdmlink")) )

921  1      {
922  2          /* variable for dealing with symm path and cross restore */
923  2          char aalName=[CDL_HOST_LENGTH];
924  2          char ssiGroup[CDL_STG_LENGTH];
925  2          char *sgdNames;    /* for returned array of STG name */
926  2          int  numGroups = 0;   /* for returned number of STG names */

928  2          /* Set up the rcmd connection (rsh method) to the client. */
929  2          int addindex = 0;
930  2          int rc = 0;
931  2          boolean_y symmpathbox = FALSE;
932  2          char *sgdNames;
933  2          int  numGroups = 0;

935  2          /* rsh connection method here */
936  2
937  2

938  1      if (! cxp->xp_have_shellcp_port)
939  2      {
940  2          struct servent *sp;

942  3          /*
943  3           * Try to get the port number again.
944  3           */
945  3          if (sp = getservbyname("shell", "tcp")) != NULL)
946  3          {
947  4              cxp->xp_shellcp_port = (ushort_c)sp->s_port;
948  4              cxp->xp_have_shellcp_port = 1;
949  3          }
950  3          else
951  3          {
952  4              pkt0.failcode = 1;
953  4              pkt0_errstr = "shell/tcp service not found";
954  4              goto send_0;
955  3          }
956  2      }

           /*
            * check for SSL enabled work item and if present add
            * SSL information to command string as an environment
            * variable--but first check for cross restore and make
            * appropriate adjustments
            */
```

```
967  2    if (pwi->ssl_groupname != NULL)
968  3    {
969  3        if (!sbc_same_host(
970  4            rcmd_stuff[0], pwi->sysname)) /* if X-recovery */
971  4        {
972  4            /* Grab the STG entries for this client */
973  4            rc = CDL_getavailablegroups(
                      rcmd_stuff[0],
                      &numgroups, &sgnames);
                  if ((numgroups==0) || (rc < 0))
                  {
976  4                char errStr[128];
977  4                sprintf(
978  5                pwi->ssl_groupname for cross restore to host %s--using
979  5                the_user_error(0, errStr);
                      network, rcmd_stuff[0];

          errStr:  'Unable to use Symwatch for cross restore to host %s--using
                                         network', rcmd_stuff[0]);
                  }
                  else
                  {
                  /*
                   * The destination client is SP enabled.
                   * First check
                   * to see if the same STG group exists
                   *                        if we don't
                   * find the same one we will use the first,
                   *                                     so set
                   * that as a default.
                   */

986  5                symPathOK = TRUE;

988  5                memset(sslgroup.0, CDL_STG_LENGTH);
989  5                strncpy(sslgroup.sgnames[0], CDL_STG_LENGTH-1);

991  5                /* Initialize the index */
992  5                addIndex = numgroups;
993  5                while (--addIndex >= 0)
994  5                {
995  5                    if (0 == strcmp(
996  5                        pwi->ssl_groupname, sgnames[addIndex]))
997  5                    {
998  5                        strncpy(
999  5                        sslgroup.sgnames[addIndex], CDL_STG_LENGTH-1);
1000 6                        continue;
1001 6                    }
1002 5                    continue;
                      }

1005 5                sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\"",
1006 5                                                sslgroup);
                          export EB_SSL_RECOVER;
                      }
                  }
                  else
                  {  /* not a cross recovery--just normal SP */
                      symPathOK = TRUE;
                      memset(sslgroup.0, CDL_STG_LENGTH);
                      strncpy(sslgroup.sgnames[0], CDL_STG_LENGTH-1);
                      sprintf(sslbuff, "EB_SSL_RECOVER=\"%s\"",
                                                      sslgroup);
                          export EB_SSL_RECOVER;
                  }
          }
```

```
1021 2    if (debugmode)
          {
              sprintf(cxp->xp_error_message,
                  "calling ELinkShell with %s, %s, %s, %s",
                  cxp->xp_shellcmd, cxp->xp_shellopt_port,
1027 3            rcmd_stuff[2], rcmd_stuff[1]);
1028 3
1029 3            the_log_state(0, "%s", cxp->xp_error_message);
                  cxp->xp_error_message[0] = 0;
          }

1030 3    strcpy(chbuf, recover_size_prefix(cxp));

1031 2    if (strlen(chbuf) > (size_t)0)
          {
1033 3        if ( symPathOK == TRUE)
              {
                  sprintf(holdbuf, "( %s %s )", chbuf, rcmd_stuff[3]);
                  rcmd_stuff[3] = holdbuf;
              }
              else
              {
                  sprintf(holdbuf, "( %s %s )", chbuf, rcmd_stuff[3]);
                  rcmd_stuff[3] = holdbuf;
              }

              /* if not Sym Path and chuf is NULL rcmd_stuff[3] stays
                                                          unchanged */
          }

          /*
          ** EDMLINK API
          */

1058 2    /* set the initial admlink transport methods */
1059      ELinkOptions = ELINK_SHELL_RCMD | ELINK_SHELL_REXEC;

1061 2    /* enable the edmlink transport method */
1062 2    if( 0 == strcmp(method, "edmlink") )
1063 2    {
1064          ELinkOptions |= ELINK_SHELL_EDMLINK;

1065 2        if( debugmode )
1066 2        {
                  /* turn on edmlink debugging */
                  ELinkOptions |= ELINK_LOGLVL_DEBUG;
              }

1072 3        /* initialize the edmlink api */
              pELinkHandle = ELinkInitAPI( ELinkOptions );
```

```c
        if( NULL == pELinkHandle ){
            sprintf(cxp->ap_error_message,
                "ERROR: Could not initialize the EXMLINK API.");

            WriteErrMsg(cxp->ap_w_prog_fd,
                    EXMLEPROGMSG_AUXPROC_ERROR, 0,
                    cxp->ap_error_message);

            rbe_log_stats(0, "%s", cxp->ap_error_message);
            return;
        }

        /* get new target host object */
        pELinkTargetObj = ELinkNewTargetObj( pELinkHandle,
                        rcmd_stuff[0] );

        if( NULL == pELinkTargetObj ){
            sprintf(cxp->ap_error_message,
                "ERROR: Could not create a new EXMLINK target
                object.");

            WriteErrMsg(cxp->ap_w_prog_fd,
                    EXMLEPROGMSG_AUXPROC_ERROR, 0,
                    cxp->ap_error_message);

            rbe_log_stats(0, "%s", cxp->ap_error_message);
            /* clean up and return */
            (void) ELinkDestroyObj( pELinkTargetObj );
            return;
        }

        /* get new user id object */
        pELinkUserIdObj = ELinkNewUserIdObj( pELinkTargetObj );

        if( NULL == pELinkUserIdObj ){
            sprintf(cxp->ap_error_message,
                "ERROR: Could not create a new EXMLINK user id
                object.\n" );

            WriteErrMsg(cxp->ap_w_prog_fd,
                    EXMLEPROGMSG_AUXPROC_ERROR, 0,
                    cxp->ap_error_message);
```

```c
            rbe_log_stats(0, "%s", cxp->ap_error_message);
            /* clean up and return */
            if( NULL != pELinkTargetObj )
                (void) ELinkDestroyObj( pELinkTargetObj );
            if( NULL != pELinkUserIdObj )
                (void) ELinkDestroyObj( pELinkUserIdObj );
                pELinkUserIdObj );

            return;
        }

        ELinkStatus = ELinkShell( pELinkHandle,
                        pELinkTargetObj,
                        pELinkUserIdObj,
                        pELinkCmdObj,
                        &rcmd_fd[0],
                        &rcmd_stderr,
                        &ELinkStatus_errno );

        rcmd_fd[1] = rcmd_fd[0];  /* this one is bi-directional */

        if (debugmode)
            rbe_log_stats(
                0, "AUXPROC: ELinkShell returned %d, errno %d,"
                " fd %d stderr fd %d\n", ELinkStatus,
                ELinkStatus_errno, rcmd_fd[0], rcmd_stderr);

        if (0 == ELinkStatus)
        {
            sprintf(cxp->ap_error_message,
                "cannot set up remote connection when calling "
                "ELinkShell with %s, %d, %s, %s, "
                "error \"%s\" (%d)",
                rcmd_stuff[0], cxp->ap_shellcp_port, rcmd_stuff[1],
                rcmd_stuff[2], rcmd_stuff[3],
                esl_strerror(
                ElinkStatus_errno), ElinkStatus_errno);

            ptcl.failcode = 2;
            ptcl.errstr = cxp->ap_error_message;

            if (debugmode)
                rbe_log_stats(
```

```
1205  3          }
1207  3          goto send_0;
1208  3        }

1210  3      /* add remainder of Symmetrix path handshaking here
1211  3       * if we have made it through the other Symm path tests
1212  3       */
1213  3      if (symmatchOK == TRUE)
1214  3      {
1215  3        int newfd;   /* fd for SSL sopen call */

1217  3      /*
1218  3       * We're going to grab the short name SSL sia alias for the
1219  3       * long network name.
1220  3       *        Useless unless this is a cross-restore.
1221  3       *
1222  3       * Because rcmd_stuff[0] = pwi->ssl_groupname on a regular
1223  3       * restore;
1224  3       */
1226  3      memset(&sslName, 0, CDL_HOST_LENGTH);
1227  3      strncpy(sslName, rcmd_stuff[0], CDL_HOST_LENGTH-1);
1228  4      if ((strlen(rcmd_stuff[0]) >= CDL_HOST_LENGTH) &&
1229  4          (0 == CDL_getsshostname(sslName, rcmd_stuff[0])))
1230  4      {
1231  4        char errStr[128];
1232  4        sprintf(
1233  4        pkt0.failcode = 2;
1234  4        pkt0.errstr = "cross-restore is configured to go
1235  4         * through SP, yet destination host is not"
1236  4         * properly configured in edm.conf";
1237  3
1238  3        goto send_0;
1239  3      }

errStr,
1240  4      /*
1241  4       * Establish the SymmPath channel
1242  4       */
1243  4      if ((newfd = CDL_client(rcmd_fd[0],
1244  4                              sslName,
1245  4                              sslGroup)) <0)
1248  4      {
1249  4        rmt_user_error(0, errStr);
1250  4        pkt0.failcode = 2;
1251  4        pkt0.errstr = "Unable to open an SSL listener connection
1252  4                      -- CDL_client failed");
1253  4        goto send_0;
1254  3      }
1255  3      /* We connected.. Use it.. */
1256  3      rcmd_fd[0] = rcmd_fd[1] = newfd ;
1257  3
1259  2      }
1261  1    }
1261  1    else if (0 == strcmp(method, "netware"))
1289  2    {
```

```
           {
1264  2      /*
1265  2       * netware connection method here
1266  2       */
1267  2      char buf[EB_MAXPATHLEN];
1269  2      char targetbuf[EB_MAXPATHLEN];
1270  2      char *tsa = "?";
1271  2      char *target = NULL;
1272  2

1276  2      filter_cmd_argv[filter_cmd_argc++] = "-/"; /* always full path */
1278  2      filter_cmd_argv[filter_cmd_argc] = NULL;

1280  2      /* Set up the rcmd connection (rsh method) to the client.
1281  2       */
1283  2      struct servent *sp;

1286  3      /* Try to get the port number again.
1287  3       */
1288  3      if (isp = getservbyname("shell", "rcp")) != NULL)
1290  3        cxp->op.shellrcp_port = (ushort_t)sp->s_port;
1291  3      else
1292  3        cxp->op.shellrcp_port = 1;
1294  4        pkt0.failcode = 1;
1295  4        pkt0.errstr = "shell/rcp service not found";
1296  4        goto send_0;
1298  3      }

1300  2      rcmd_fd = rcmd_fds; /* set valid pointer */

1305  2      /* now skip over ./host/bin/starter
1306  2       */
1307  2      for (p2 = rcmd_stuff[3]; '\0' != p2 && ' ' != *p2; p2++)
1309  2      { /* skip ./host/bin/starter */
1310  1
1311  2      }
1312  1      /* if a -c target; then take out target;
1313  2       */
1314  1      while (' ' == *p2)
1315  2        p2++; /* advance to start of next word */

1317  2      if (('-' == *p2) && ('c' == p2[1]))
1319  2      {
1320  2        *targetbuf = '\0';
1321  2        if (strlen(p2) >2)
1323  2        {
1324  3          char *p3 = p2+2;
```

```c
        char *p4;
        while (' ' == *p3)
        {
            p3++; /* advance to start of next word */
        }
        p4 = p3; /* save this as start of the prefix */
        for (; '\0' != *p3; p3++)
        {
            if (((':' == *p3) && (':' == p3[1])) ||
                ((':' == *p3) && ('\\' == p3[1])))
            {
                p3++;
                break;
            }
        }
        if (*p3 != 0) /* if found the prefix */
        {
            char * tempChar = NULL;
            strncpy(targetbuf, p4, p3-p4);
            targetbuf[p3 - p4] = '\0';
            target = targetbuf;
            if (':' == p3[1])
            {
                p3++;
                tempChar = strchr(p3+1, '\\');
                if (NULL != tempChar)
                {
                    tempChar[0] = ' ';
                    strcpy(p4, p3+1);
                }
            }
        }
    }

    /*
     * now, if it's a netware cross recovery, get password,
     * destination target -- don't use source target from pw.
     * pwi points to work item config struct backup was done under
     */

    if (0 != strcmp(rmd_stuff[0], pwi->sysname))  /* if x-recovery */
    {
        RBC_WORKITEM *pwi2 = pwi;

        tsa = "";
        if (*targetbuf != 0)
        {
            target = targetbuf;
        }

        for (pwg = cmc->cmc_config->bc_config; NULL != pwg; pwg = pwg->next)
        {
            for (pwi = pwg->item; NULL != pwi; pwi = pwi->next)
            {
                if (abc_same_host(pwi->sysname, rmd_stuff[0]))
                {
                    goto goti2;
                }
            }
        }
    }
goti2:;
}
```

```c
    /*
     * if we did not find a wi for the target system,
     * put user prompted here.
     */

    if (NULL == pwi)
    {
        pwi = pwi2; /* restore old value if i found */
    }

    /*
     * check for an encrypted password
     */

    if (NULL != pwi && (pwi->flags & WORKITEM_FLAGS_ENCRYPT_PASSWD))
    {
        pwi->flags &= ~(WORKITEM_FLAGS_ENCRYPT_PASSWD)
    }

    if (NULL != pwi)
    {
        sprintf(buf9, "%s %s -target %s -tsa %s -login %s -opassword %s", p2,
            tsa,
            (target != NULL) ? target : "",
            (target != NULL) ? tsa : "",
            pwi->nw_clnt_target != NULL ? pwi->nw_clnt_target : "",
            pwi->nw_username != NULL ? pwi->nw_username : "",
            pwi->nw_passwd != NULL ? pwi->nw_passwd : "");
    }
    else
    {
        sprintf(buf9, "%s %s -target %s -tsa %s -login %s -password %s", p2,
            tsa,
            (target != NULL) ? target : "",
            (target != NULL) ? tsa : "",
            pwi->nw_clnt_target != NULL ? pwi->nw_clnt_target : "",
            pwi->nw_username != NULL ? pwi->nw_username : "",
            pwi->nw_passwd != NULL ? pwi->nw_passwd : "");
    }

    if (NULL != pwi && (CNCIN_NWBARE != pwi->connection_type))
    {
        if (CNCIN_NWBARE != pwi->connection_type)
        {
            WriteEventStringMsg(cep->cep_v_prog_fd,
                EDMRERPROGMSG_AUXPROC_WARNING, 0,
                "work item '%s' specifies "
                "connection method '%s' but client "
                "was installed to use the netware "
                "method -- using netware method",
                pwi->name,
                CNCIN_RSH == pwi->connection_type ? "Rsh" :
                CNCIN_EDMLINK == pwi->connection_type ? "EdmLink" :
                CNCIN_DIRECT == pwi->connection_type ? "Direct" :
                CNCIN_SOCKET == pwi->connection_type ? "Socket" :
                CNCIN_NETWARE == pwi->connection_type ? "Netware" :
                "????");
        }
    }
```

```
1441  3          if (0 != pwi->connection_port) {   /* if port specified */
1442  4              rbe_log_state(
1443  5                  0, "using socket port %d to connect to"
1444  5                  " host \'%s\', witem",
1445  4                  pwi->connection_port,
1446  5                  pwi->name);
1447  5          }

1449  4          cxp->ap_shellrcp_port = (ushort_t)pwi->connection_port;

1450  4      }

1452  3      if (debugmode)
1453  3          rbe_log_state(
1455  2              0, "calling nxrcmd with %s, %s, %d, %d, %s\n",
1456  3              rcmd_stuff[0],
1457  3              rcmd_stuff[1], cxp->ap_shellrcp_port,
1458  3              rcmd_stuff[2], buf9);

1460  3      rcmd_fd[0] = nxrcmd(&rcmd_stuff[0], cxp->ap_shellrcp_port,
1461  3          rcmd_stuff[1],
1462  2          rcmd_stuff[1], rcmd_stuff[2], buf9);

1464  2      /* this one is bi-directional */

1467  2      if (debugmode)
1468  2          rbe_log_state(
1469  2              0, "AUXPROC: nxrcmd returned fd %d, stderr fd %d\n",
1470  2              rcmd_fd[0], rcmd_stderr);

1471  2      rcmd_fd[1] = rcmd_fd[0];    /* this one is bi-directional */

1473  2      if (rcmd_fd[0] == -1)
1474  2      {
1475  2          pkt0.failcode = 2;
1476  3          pkt0.errstr = "cannot set up remote connection",

1478  3          goto send_0;

1479  3      }
1480  3      else if (0 == strcmp(method, "socket"))
1481  3      {
1482  3          /* socket connection method here */
1484  2      }

1485  2      if (CNCTN_SOCKET != pwi->connection_type)  /* if other than rsh */
1486  2      {
1487  2          WriteMethodString(cxp->ap_w_prog_fd,
1488  2              EDMHSPROCMSG_AUXPROC_WARNING, 0,
1489  4              "work item \'%s\' specifies",
1490  4              "connection method %s but",
1491  4              "client was intended to use the",
1492  4              "socket method -- using socket method",
1493  4              pwi->name,
1494  4              pwi->name,
1495  4              method);
1496  4      }

1497  4      CNCTN_RSH == pwi->connection_type ? "Rsh" :
```

```
1498  4          CNCTN_EDMLINK == pwi->connection_type ? "Edmlink" :
1499  6          CNCTN_DIRECT == pwi->connection_type ? "Direct" :
1500  4          (
1501  4          (
1502  4          CNCTN_SOCKET == pwi->connection_type ? "socket" :
1503  4          (
1504  4          "???")))));
1505  3      }

1507  4      CNCTN_NETWARE == pwi->connection_type ? "Neware" :
1508  5
1509  5          "???")))));

1510  3      /* read file with name of wi ( */
1511  3      /* written by listener() to get info about client */

1513  4      socket_port = -1;
1514  4      debugno = 0;
1515  5      strcpy(wifilename, wifiledir);
1516  5      strcat(wifilename, socket_file);
1517  3      if (NULL == socket_file)
1518  3      {
1519  3          rbe_log_state(
1520  3              0, "No socket file name passed to auxproc\n");
1521  4          return;
1522  5      }

1528  3      /* Get socket information from eblistend info file */
1529  4      errno = eb_parse_listener_info_file (wifilename,
1530  3          buf,
1531  4          &socket_host,
1534  4          &debugno,
1535  4          &socket_port,
1536  1          listener_filename);

1538  3      if (errno != E_SUCCESS)
1539  4      {
1540  4          /* The routine already logged an error message, simply */
1542  4          /* return the error to caller */
1543  4          return;
1544  4      }

1546  4      rbe_log_state(0,
1547  4          "Unable to read eblistend info file \'%s\'\n",
1548  5          wifilename);

1549  5      if (socket_port == -1)
1550  5      {
1551  5          rbe_log_state(
1552  5              0, "Invalid port field seen for client\n");
1554  5          return;
1555  5      }

            if (0 != pwi->connection_port)   /* if port specified */
            {
                rbe_log_state(
                    0, "using socket port %d to connect to"
                    " host \'%s\', witem",
                    pwi->connection_port,
                    pwi->name);
```

```c
        }
        socket_port = pwi->connection_port;
    }

    if (!debugmode)
        tbe_log_state(
            0, "host= %s port=%d seq=%d\n", socket_host, socket_port, dbeugno);

    /* We need to fix this up */

    rcmd_fd = rcmd_fds; /* set valid pointer */

    /* set up socket connection to call the right type
     * of socket (SYN) */

    rcmd_fd[0] = sopen(socket_host, 'c', socket_port);
    if (rcmd_fd[0] < 0)
    {
        tbe_log_state(0, "Socket Open error: %d\n", rcmd_fd[0]);
        _exit(3);
    }

    /*
     * set up ONLY remote command file descriptors to client
     * via socket connection. DO NOT overwrite filter_cmd fd's
     */

    rcmd_fd[1] = dup(rcmd_fd[0]);

    /* If we are connected, determine if the user wants normal sockets
     * or ssl.
     */

    if (rcmd_fd[0] != -1)
    {
        if (!pwi->ssl_groupname)
        {
            /*
             * send OK status to client waiting on data socket
             * before
             * data stream
             */

            sprintf(chuf, "OK: %d socket connection made;
                ", dbeugno);
            (void)dwrite (rcmd_fd[0], chuf, (int)strlen(chuf));
        }
        else /* ssl connect */
        {
            int numgroups = 0;
            char ssName[CDL_HOST_LENGTH];
            char ssGroup[CDL_STG_LENGTH];
            char *scgNames = 0;
            int addindex = 0;
            int rc = 0;

            /* First test to see if this is a cross restore. If
             * so we can't just use the source STG group from the
             * client
```

```c
             * since it may not exist on the destination client.
             * We need to see if the destination is SP enabled and
             * get a good STG group for that.
             *                  We'll use the same group
             * if it is available.  If no STG group is available we will
             * fall back to network.
             */

            if (!sbc_same_host)
                rc = CDL_getavailablesrggroup(); /* if X-recovery */

            /* Grab the STG entries for this client */
            if (numgroups==0 || (rc < 0))
            {
                char errStr[128] ;
                sprintf(
                    errStr, "Unable to use Symbohx for cross restore to host %s-using network", socket_host);

                the_user_error(0, errStr);

                /*
                 * Default to network and send normal handshake.
                 */

                /*
                 * send OK status to client waiting on data socket
                 * before
                 */

                sprintf(chuf, "OK: %d socket connection made;
                    ", dbeugno);
                (void)dwrite (rcmd_fd[0], chuf, (int)strlen(chuf));
            }
            else
            {
                /*
                 * The destination client is SP enabled.
                 * First check
                 * to see if the same STG group exists. If we don't
                 * find the same one we will use the first, so set
                 *
                 * that as a default
                 */

                memset(ssGroup, 0, CDL_STG_LENGTH);
                strncpy(
                    ssGroup, scgNames[0], CDL_STG_LENGTH-1);

                /* Initialize the index */
                addindex = numgroups;
                while (0 == strcmp(
                    pwi->ssl_groupname, scgNames[addindex]))

                    strncpy(
                        ssGroup, scgNames[addindex], CDL_STG_LENGTH-1);
                    continue;
                }

                /*
                 * We need to get a short name SSL alias if
                 * the destination
```

```
        /* name happens to be long
         */
        memset(sslName, 0, CDL_HOST_LENGTH);
        strncpy(
            sslName, socket_host, CDL_HOST_LENGTH-1);
        if (strlen(
            socket_host) >= CDL_HOST_LENGTH &&
            (0 >= CDL_gethostname(sslName, socket_host)))
        {
            char errStr[128] ;
            sprintf(
                errStr, "Unable to determine the short SymmPath hostname of %s",
                socket_host);
            the_user_error(ItEd, errnum, errStr);
            pkto.failcode = 2;
            pkto_errstr = "cross-restore is "
            " go through SP, yet desitation host is not"
            " properly configured in adm.conf",
            configured in adm.conf
            goto send_0;
        }
        if (ebsock_server_connect_ssl(sslGroup,
                socket,
                &cmd_fd[0],
                &cmd_fd[1],
                socket_host,
                holdbuf) < 0)
        {
            WriteRmtStringMsg(cxp->xp_w_prog_fd,
                EUMREPROCMSG_AUXPROC_ERROR, 0,
                "%s", holdbuf);
            return;
        }
        /*
         * Just a normal $p restore--nothing special to do
         */
    }
    else
    {
        if (ebsock_server_connect_ssl(sslGroup,
                pwi->ssl_clientauthname,
                &cmd_fd[0],
                &cmd_fd[1],
                socket_host,
                holdbuf) < 0)
        {
            WriteRmtStringMsg(cxp->xp_w_prog_fd,
                EUMREPROCMSG_AUXPROC_ERROR,
                0,
                "%s", holdbuf);
            return;
        }
```

```
        /* Now that we are finally connected, dup to std err */
        rcmd_stderr = CDL_dup(rcmd_fd[0]);
    }
    else
    {
        /*
         * unknown connection method here
         */
        WriteRmtStringMsg(
            cxp->xp_w_prog_fd, EUMREPROCMSG_AUXPROC_ERROR, 0,
            "bad connection method \"%s\"\n", method);
        return;
    }

    if (0 == pipe(xcpiopen_pipe))
    {
        WriteRmtStringMsg(
            cxp->xp_w_prog_fd, EUMREPROCMSG_AUXPROC_ERROR, 0,
            "could not set up progress channel from "
            "xcpiopen.\n");
        _exit(2);
    }

    switch (xcpiopen_pid = fork())
    {
    case -1:
        /* error */
        (void) close(xcpiopen_pipe[0]);
        (void) close(xcpiopen_pipe[1]);
        pkto.failcode = 2;
        pkto_errstr = "fork failed";
        pkto.errnum = errno;
        break;

    case 0: /* child */
    {
        /*
         * Make our stdin be the bulk-from-parent stream.
         */
        (void)dup2(xcpiopen_pipe[0], 0);

        if (debugmode)
        {
            the_log_state(
                0, "FILTER CMD: \"%s\", human_name: \"%s\"\n",
                filter_cmd, human_name);
            for (i = 0; i < filter_cmd_argc; i++)
                the_log_state(0, "\t%s\n", filter_cmd_argv[i]);
        }

        /*
         * If no fd info passed to child, then child stdout goes to the
         * rcmd process. Otherwise, arbitrary fd is used and we pass
         * the fd number as an argument per the format given to us
         */
        if (filter_cmd_fd_info == -1)
```

```c
1771  3          filter_cmd_fdout = 0;
1772  3          (void)dup2(rcmd_fd[0], 1);
1773  3      }
1774  3      else
1775  3      {
1776  3          filter_cmd_fdout = rcmd_fd[0];
1777  3          (void) sprintf(
1778  3              fdarg, filter_cmd_fd_info,
1779  3              filter_cmd_fdout);
1780  3          filter_cmd_argv[filter_cmd_fd_info] = fdarg;
1781  3      }
1782  3
1783  3      /*
1784  3       * Send real uid string on filter_cmd_fdout to client process
1785  3       */
1786  2      i = (int)strlen(human_name);
1787  2      if ((n = looppw(
1788  2          filter_cmd_fdout, human_name, i, write_CDL_no_eintr)) != i)
1789  3      {
1790  3          rbe_log_stats(RBRECOVER_MKERR(
1791  3              errno), "Can't write %d bytes to host \"%s\", "
1792  3              "ec=%d, errno=%d, id=%d\n", i, rcmd_stuff[0],
1793  3              n, errno);
1794  3
1795  3          _exit(3);
1796  3      }
1797  1
1798  2      _exit(2);
1799  3
1800  3      rbe_log_stats(RBRECOVER_MKERR(
1801  3          errno), "Can't write %d bytes to host \"%s\", "
1802  3          "ec=%d, errno=%d, id=%d\n", i, rcmd_stuff[0],
1803  3          n, errno);
1804  3
1805  2      _exit(1);
1806  2      (void)close(xcpiogen_pipe[0]);
1807  2      (void)close(xcpiogen_pipe[1]);
1808  2
1809  2      /* Call the CDL layer so we can warn an SSL, that an execvp
1810  2       * is coming. If this isn't an SSL socket this call is a
1811  2       * no-op.
1812  2       */
1813  2      (void)CDL_execvpPrep(filter_cmd_fdout);
1814  2      (void)execvp(filter_cmd, filter_cmd_argv); /* run xcpiogen */
1815  2      rbe_log_stats(RBRECOVER_MKERR(
1816  2          errno), "Can't exec \"%s\", errno=%d\n",
1817  2          filter_cmd, errno);
1818  2      _exit(1);
1819       break;
1820
1821       default:
1822  2      /* parent */
1823  2      (void)close(xcpiogen_pipe[1]);
1824  2      xcpiogen_prog_fd = xcpiogen_pipe[0];
1825  2      break;
1826       }  /* end of switch */
1827
1828       /* Parent has no use for this file descriptor any more,
1829  1        and making it
```

```c
1830  1       * now is important so that if the child dies while the rcmd still
1831  1       * wants more input from the child, the rcmd will die too
1832  1       *                                                    instead of
1833  1       * hanging around hoping that maybe this parent process will supply
1834  1       * the data  */
1835
1836  2      if (ipv1->ssl_groupname)
1837  2      {
1838  2          if (rcmd_fd[0] != -1)
1839  3          {
1840  1              (void)close(rcmd_fd[0]);
1841  1              if (rcmd_fd[1] != -1)
1842  2                  (void)close(rcmd_fd[1]);
1843  3          }
1844  3      }
1845       else
1846       {
1847  2          /* we didn't really dup these if they were a network
1848  3           * avoid the closes in that case
1849  3           */
1850  3          if (rcmd_fd[0] != -1)
1851  3              (void)close(rcmd_fd[0]);
1852  1
1853  1          /* However in the case of rarcmd we did get a network
1854  1           * socket for rcmd_stderr so we do need to issue one
1855  1           * CDL_close on the one SSP socket. We can detect that
1856  2           * case because rcmd_stderr will not be equal to rcmd_fd[0]
1857  3           */
1858  3          if (rcmd_stderr != rcmd_fd[0])
1859  3          {
1860  4              if ( 0 == CDL_isSSLsocket(rcmd_fd[0]) )
1861  4                  (void) CDL_close(rcmd_fd[0]);
1862  4              /*
1863  4               * Not a SSL socket. do the close if not
1864  4               * we close SP socket in XCPIOGEN after
1865  4               * data has been moved by calling SPxexit()
1866  4               */
1867  4          }
1868  5      }
1869  5
1870  4
1871  4      rcmd_fd[0] = -1;
1872  1
1873  1  }
1874  1
1875
1876  1      if ((0 == strcmp(method, "rsh")) || (0 == strcmp(
1877  2          method, "admlink")) )
1878  2      {
1879  2          /* Because of occasional loss of connection, due to timing issues when
1880  2           * connected via admlink,
1881  2           * turn on keepalive. In version 4.7 this will
1882  3           * also be addressed by admlink directly. (OSGsw38596)
1883  3           */
1884  2          int on = 1;
1885  2          if (-1 == setsockopt( rcmd_stderr,
1886  2                          SOL_SOCKET,
1887  2                          SO_KEEPALIVE,
1888  2                          (char *)&on,
1889  2                          sizeof(on) ))
1890  3          {
1891           rbe_log_stats( RBRECOVER_MKERR(errno),
1892               "Warning: setsockopt for SO_KEEPALIVE
1893               failed" );
```

```c
1891  1      }
1892  1
1894  2      if(0 == pkt0.failcode)
1895  1      {
1896  1          /* Lets check here to see if xpciogen is still running.
1897  2           * The child could have exited prior to the fork.
1898  2           */
1899  2          int  childone_ret;
1900  2          int  childbxitstatus;
1902  3
1903         xpciogen_still_running = TRUE;
1904  2      childone_ret = childone(xpciogen_pid, &childbxitstatus);
1905  2      if(-1 == childone_ret)
1906  3      {
1907  3          the_log_stats(HBRECOVER_MKOR(errno),
1908  3              "internal error: testing to see if the fork
1909                   exited."),
1910         }
1911  2      else if (0 != childbxitstatus)
1913  2      {
1914  2          xpciogen_still_running = FALSE;
1915  2          r_resultdata = childbxitstatus;
1916         }
1917         }
1918
1919  1      r_resultdata = 0;
1920  1
1921  1      /*
1922          * Send the setup failure/success packet.
1923          */
1924  1  send_0:
1925  1      if (pkt0.msglen < 0)
1926  2      {
1928  2          pkt0.msglen = (int)strlen(pkt0_errstr) + 1;
1929         }
1930  2
1931  2      c = '\0';
1931  1      c += (int)sizeof(pkt0) + pkt0.msglen;
1932  2      pwrite_or_die(cop->ap_w_fd, &c, 1, _exit);
1933  2      pwrite_or_die(cop->ap_w_fd, (char *)&i, sizeof i, _exit);
1934  2      pwrite_or_die(cop->ap_w_fd, (char *)&pkt0, sizeof pkt0, _exit);
1938  1      if (pkt0.msglen > 0)
1939  3      {
1939          sprintf(cop->ap_error_message, "%s", pkt0_errstr);
1940  3          the_log_stats(cop->ap_error_message);
1941  3          pwrite_or_die(cop->ap_w_fd, pkt0_errstr, pkt0.msglen, _exit);
1941  1      }
1943
1944  1      /* If the fork was successful,
1944          *   wait for the remote exit status to come back
1949          * on stderr,
1948          *   and send it to parent in an 'R' reply.  If the fork was
1947          * unsuccessful, there is no 'R' reply, and, furthermore, the value in the
1950          * 'r' reply is meaningless.
1952          */
1953  1      if ((xpciogen_pid > 0) &&
1953          (TRUE == xpciogen_still_running))
1956  2      {
1954          int remote_exitcode = 0;
1955          int remote_exitinfo;
1957          Demux_ret = DemuxAuxchildren(cop->ap_w_prog_fd,
```

```c
                    send_stderr,
                    rcmd_stuff_fd,
                    xpciogen_prog_fd,
                    xpciogen_pid,
                    &remote_exitinfo);

            if(0 != Demux_ret)
            {
                /*
                 * Error: monitoring Auxproc's children. */;} _exit);
            }

            /* error_logging */

            /*
             * notify the main program of remote success/failure
             */

#if 0
            /*
             * If our connection method anything BUT "netware"?
             */
            if (0 == strcmp(method, "netware"))
            {
                /*** YES ***/
                /* Now wait for the exit code of the filter (local).
                 */

                /*
                 * waitpid() may return with EINTR before the child
                 * proc exits, in that case, we want to continue
                 * with the wait. (fix for OSSW01547)
                 */
                while (waitpid(xpciogen_pid, &wait_result, 0) == -1) &&
                       (EINTR == errno))
                {
                    /* empty while loop */
                }

                /*
                 * exited while loop either because the return value of
                 * waitpid is NOT -1, or the errno is NOT EINTR
                 */
            }
            else
            {
                /*** IT'S NETWARE ***/
            }
            rbe_log_stats(0, "Auxproc(%d) remote exit is %s.", getppid(),
                remote_exitinfo);
            pwrite_or_die(cop->ap_w_fd, &c, 1, _exit);
            pwrite_or_die(cop->ap_w_fd, (char *)&i, sizeof i, _exit);
            pwrite_or_die(cop->ap_w_fd, (char *)&remote_exitinfo, i, _exit);
#endif

            /*
             * This is the main program of remote children.";} _exit);
             */
        }
    }
```

```
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
**** HACK **** HACK **** HACK **** HACK
```

```
/*
 * this hack is basically for Netware.
 *
 *                              Netware TCP/IP close()
 * does not send anything back to the (
 *                                     localfilter process
 * So the 'filter' process
 * that is running as pid pktd_pid.
 *                                 there isn't
 * just keeps on sending data across the network and
 *
 * anybody there to read it.
 *
 * Therefore,
 *     this 'suoproc' doesn't finish until all data has
 *     been written to the socket even though no process is
 *                                                     reading
 * it. However,
 *     the process might hang indefinitely if the network
 *     buffer(s) become(s) full.
 *
 * this is how we will get around this (
 *                                      kind of a back but could
 * not think of a better way.  None of the normal TCP/IP
 * mechanisms seem to work with Netware.
 *
 * If the exitinfo
 *     which is zero for success or non-zero for
 * failure) reports failure...
 *
 * After we have received the exit info from the Netware
 *                                                   client
 * then try to get it's "wait_result" twice with a
 *                                            second pause
 * in between.  If we don't get it,
 *                            we have to assume that the
 * process is hung and can't wind down.
 *
 * Therefore,
 *     let's send a SIGPIPE to the process.  I chose
 * SIGPIPE because it is usually trapped by our
 *                                        software and
 * handled in a controlled manner.
 *                            Just keep on trying to kill
 * it, sleeping for 2 seconds.
 *                     and then see if it went away until
 * it DOES go away.
 *
 * Just keep on trying to kill it,
 *                        sleeping for 2 seconds. And
 * then see if it went away until it DOES go away.
 *
 * Apparently,
 *     the kill)
 * SIGPIPE) is ignored by the process while
 * there is activity occuring like positioning the tape.
 *                                                   That is
 * the reason for having to do the kill(
 *                                      ) in a loop until it
 * works.
 */

if (remote_exitinfo)
{
   /*
    ** We recieved an error exit status from Novell
    */
   int exitinfo;
```

```
   exitinfo = waitpid(
                      xcplogen_pid, &wait_result, WNOHANG);
   while (!exitinfo)
   {
      /*
       * Wait a couple of seconds longer and try again
       */
      sleep(2);
      exitinfo = waitpid(
                         xcplogen_pid, &wait_result, WNOHANG);
   }

   if (!exitinfo)
   {
      kill(xcplogen_pid, SIGPIPE);    /*  BLAST IT! */
   }

   /**
    ** main command loop in caller will send this result for us
    */

}
else
{
   /*
    ** We received a success return status from Novell
    */

   /*
    * waitpid() may return with EINTR before the child
    * proc exits. In that case, we want to continue
    * with the wait. (fix for OSRev#15687)
    */
   while ((waitpid(
                  xcplogen_pid, &wait_result, 0) == -1) &&
          (EINTR == errno))
   {
      /* empty while loop */
   }

   /* exited while loop either because the return value
    * of
    * waitpids NOT -1, or the errno is NOT EINTR
    */

   if (WIFEXITED(
                 wait_result))
   {
      /*
       * wait_result contains the exit status of the child
       * process. Use the standard macros to determine the
       * status of the child process and set the return value
       * to indicate the status accurately.
       */

      /*
       * set the result to the exit code
       */
      wait_result = WEXITSTATUS(wait_result);
   }
```

```
2116  2          else if (WIFSIGNALED(wait_result))
2117  3          {
2118  3              /*
2119  3               * child proc exited due to an uncaught signal, so
2120  3               * set the exit code to the signal number plus our own
2121  3               * code XC_EXIT_SIGBASE.
2122  3               */
2123  3
2124  3              wait_result = WTERMSIG(wait_result) + XC_EXIT_SIGBASE;
2125  3          }
2126  3          else if (WIFSTOPPED(wait_result))
2127  3          {
2128  3              /*
2129  3               * child proc is stopped, set the exit code accordingly.
2130  3               */
2131  3
2132  3              wait_result = XC_EXIT_STOPPED;
2133  3          }
2134  2      }
2135  2
2136  1      r_resultdata = wait_result;
2137  1  }
2138  1
2139  1  if (rcmd_stderr != -1)
2140  2  {
2141  2      /* OSGsw34952 -- Workaround for SFPclose() behavioral defect
2142  2
2143  2       * Shut off the SSL (SFP) exit handler
2144  2       * If not a SSL socket, do the close if not
2145  2       * we close SP socket in XCFTDGEN after
2146  2       * data has been moved by calling SFPexit()
2147  2       */
2148  2      if ( 0 == rcmd_isSSLsocket(rcmd_stderr) )
2149  3      {
2150  3          (void)close(rcmd_stderr);
2151  3          rcmd_stderr = -1;
2152  3      }
2153  2      else /* SSL socket used as stderr, so clean up
2154  2              the SP stuff */
2155  2      {
2156  2          /*
2157  2           * SP socket was set to stderr of remote
2158  2           * commands. Since we call CDL_rootexit
2159  2           * when we created the SP socket we will
2160  2           * explicitly call CDL_exit() that will
2161  2           * clean up the entire SP environment. */
2162  2
2163  2          if ( 0 == sp_cdlexitdone )
2164  3          {
2165  3              CDL_exit();
2166  3              sp_cdlexitdone = 1;
2167  3          }
2168  2      }
2169  1  }
2170  1
2171  1  /*
2172  1   * set up variables that main loop will use to send 'r' reply
2173  1   */
2174  1  cxp->ap_resultlen = sizeof r_resultdata;
2175  1  cxp->ap_resultdata = (char *)&r_resultdata;
      }
      /* end of z_rcmdfilter() */
```

```
2178  static enum input_states decodecookie(char *cp);

2181  struct cookie2state {
2182      char *cookie;
2183      enum input_states state;
2184  };

2186  static struct cookie2state c2stbl[] = {
2187  1      { REMFD_COOKIE_PIDOUT,
                                          INSTATE_COPY_TO_STDOUT ),
2188  1      { REMFD_COOKIE_PIDOUT2,
                                          INSTATE_COPY_TO_STDOUT ),
2189  1      { REMFD_COOKIE_P2OUTEND,      INSTATE_SEARCH_PREFIX0 ),
2190  1      { REMFD_COOKIE_STATUS,        INSTATE_GATHER_STATUS ),
2191  1      { NULL,                       INSTATE_NO }
2193  1  };

2195  static enum input_states
2196  decodecookie(char *cp)
2197  {
2198      struct cookie2state *c2s;

2201  1      for (c2s = c2stbl; c2s->cookie != NULL; c2s++)
2202  2      {
2203  2          if (strcmp(c2s->cookie, cp) == 0)
2204  3          {
2205  3              return c2s->state;
2206  2          }
2207  2      }
2208  1      return INSTATE_SEARCH_PREFIX0;
2209  1  }   /* end of decodecookie() */
```

```
2211   static void
2212   siguser1_handler(int sigvalue)
2213 1 {
2214 2     if (debugmode)
2215 2     {
2216 2         rbe_log_stats(0,"\nUSR1 (ATTN) signal received!\n");
2217 2     }
2219 1     if (0 < xcpiopen_pid)
2220 2     {
2221 2         (void)kill(xcpiopen_pid, SIGTERM);
2222 1     }
2224 1     +siltrl_ecl;
2225   }   /* end of siguser1_handler() */
```

```
2227   static void
2228   sigterm_handler(int sigvalue)
2229 1 {
2230 2     if (debugmode)
2231 2     {
2232 2         rbe_log_stats(0,"\nTERM signal received!\n");
2233 1     }
2235 1     rbe_close_logs(logging_channel);
2237 1     /* BSS workforgroup: Make a call to CDL_exit
2238 1        if symptom/b workitem and CDL_exit has
2239 1        not already been called.  This is to clean
2240 1        up sockets since we have made a call to
2241 1        CDL_noatexit, we have to make sure we
2242 1        clean up */
2244 1     if ( (1==is_symptomb) && (0==sp_cdlexitdone) )
2245 2     {
2246 2         CDL_exit();
2247 2         sp_cdlexitdone = 1;
2248 1     }
2250 1     signal(SIGTERM, SIG_DFL);  /* restore default (terminate) action */
2251 1     (void)kill(getpid(), SIGTERM);
2253   }   /* end of sigterm_handler() */
```

```
2256      /*
2257       * Invoked when running recover in debugging mode.
2258       * This exec's a separate a.out file to implement the auxproc, so that
2259       * breakpoints can be set in the recover code without affecting
2260       * the auxproc code.
2261       */
2262
2263      static void
2264      z_exec_separate_auxproc(struct auxproc_context *cxp,
2265                              char *pathname)
2266      {
2267          char procnum_str[32];
2268          char r_fd_str[32];
2269          char w_fd_str[32];
2270          char r_bulk_fd_str[32];
2271          char dbgmodestr[32];
2272          char *argv0;
2273
2274          /*
2275           * NOTE: This name is "special"; the recover main()
2276           * function looks for it to know when it's being invoked
2277           * to perform auxproc processing.
2278           */
2280          argv0 = "ebr_auxproc";
2282    1:    (void)sprintf(procnum_str, "%d", cxp->cxp_my_auxnum);
2283    1:    (void)sprintf(r_fd_str, "%d", cxp->cxp_r_fd);
2284    1:    (void)sprintf(w_fd_str, "%d", cxp->cxp_w_fd);
2285    1:    (void)sprintf(r_bulk_fd_str, "%d", cxp->cxp_r_bulk_fd);
2286    1:    (void)sprintf(dbgmodestr, "%d", debugmode);
2287    1:    (void)execlp(pathname,      /* prog to execute */
2289    1:             argv0,             /* argv 0 */
2290    1:             procnum_str,       /* argv 1 */
2291    1:             r_fd_str,          /* argv 2 */
2292    1:             w_fd_str,          /* argv 3 */
2293    1:             r_bulk_fd_str,     /* argv 4 */
2294    1:             dbgmodestr,        /* argv 5 */
2296    1:             (char *)0);        /* end-of-args */
2297
2298    1:    /* If we get here, the exec did not go. Caller will bomb for us. */
2300      }                         /* end of z_exec_separate_auxproc() */
```

```
2303      /*
2304       * Use these functions with loop_rw to build
2305       * loop_read (or loop_write) that ignores EINTR.
2306       */
2308    2: static int
2309    2: read_CDL_no_eintr(int fd,
2310    2:                   char *buf,
2311    2:                   int nbytes)
2312    1: {
2313    1:     int r;
2315    1:     do {
2316    2:         errno = 0;
2317    2:         r = CDL_read(fd, buf, (uint_t)nbytes, 0);
2318    1:     } while (r == -1 && errno == EINTR);
2319
2331    1:     return r;
2332    1: }                         /* end of read_CDL_no_eintr() */
```

```
static int
write_CDL_no_eintr(int fd,
                   char *buf,
                   int nbytes)
{
    int r;

    do
    {
        errno = 0;
        r = CDL_write(fd, buf, (uint_t)nbytes, 0);
    } while (r == -1 && errno == EINTR);

    return r;
    /* end of write_CDL_no_eintr() */
}
```

```
/*
 * Wait, interruptibly, for at least one byte to become available on fd.
 *
 * Returns 0 if at least one byte is available.
 * Returns -1 for any type of failure including wait interruption.
 * Sets errno appropriately when -1 is returned.
 */

int
fd_avail_1_wait_intr(int fd)
{
    fd_set_ty rdbits;

    FD_ZERO(&rdbits);
    FD_SET(fd, &rdbits);

    /*
     * Don't need to examine rdbits after select, since only one
     * fd is in the set -- therefore return value can be computed
     * directly from select return value.
     */
    if (sel_select(E_FD_SETSIZE, &rdbits, NULL, NULL, NULL) == -1)
    {
        return -1;
    }
    return 0;
    /* end of fd_avail_1_wait_intr() */
}
```

```
2372    /*
2373     * Test, interruptibly,
2374     *      for at least one byte to become available on fd.
2375     *
2376     * Returns 1 if at least one byte is available.
2377     * Returns -1 for any type of failure, including test interruption.
2378     * Return 0 if no data is available.
2379     *
2380     * Sets errno appropriate when -1 is returned.
2381     */
2382    int
2383    fd_avail_test_intr(int fd)
2384  1 {
2385        int retStatus;
2386  1     struct pollfd fd_test;
2387
2388        fd_test.fd = fd;
2389  1     fd_test.events = POLLIN;
2390  1     fd_test.revents = 0;
2391  1
2392        /*
2393         * Don't need to examine rdbits after select, since only one
2394  1      * fd is in the set -- therefore return value can be computed
2395  1      * directly from select return value.
2396  1      */
2397  1
2398  1     if ((retStatus = CDL_poll_read(&fd_test, 1, 0)) == -1)
2399        {
2400  1         /* ERROR encountered */
2401  2         return -1;
2402  2     }
2403  1
2404  1     return retStatus;
2405  1
2407    }   /* end of fd_avail_test_intr() */
```

```
2409    /*
2410     * Test, for at least one byte to become available on fd.
2411     *
2412     * Returns 1 if at least one byte is available.
2413     * Returns -1 for any type of failure other than EINTR.
2414     * Return 0 if no data is available.
2415     *
2416     * Sets errno appropriate when -1 is returned.
2417     */
2418    int fd_avail_test(int fd)
2419  1 {
2420        int retStatus;
2421        struct pollfd fd_test;
2422  1
2423        fd_test.fd = fd;
2424        fd_test.events = POLLIN;
2425  1     fd_test.revents = 0;
2426  1
2427        /*
2428         * Don't need to examine rdbits after select, since only one
2429  1      * fd is in the set -- therefore return value can be computed
2430  1      * directly from select return value.
2431  1      */
2432  1
2433  1     while ((-1 == (retStatus = CDL_poll_read(&fd_test, 1, 0))) &&
2434  2            (EINTR == errno))
2435  1     {
2436  2         ;
2437  2     }
2438  1
2439  1     return retStatus;
2440  1
2441    }
```

```c
#define EBR_FORK fork    /* portability definition */

static int ebr_direct_rcmd_fds[2] = { -1, -1 };

/*
 * Function to do a direct fork()/exec/
 *
 * client programs put to in & out fds for stdio w/ child process.
 * Returns pid to fds if successful, NULL if an error.  stderr from new
 * process is returned on *fd2p.
 *                 Should only be called when client = server.
 */
int
ebr_direct_rcmd(char        **ehost,
                ushort      iipport,
                struct auxproc_context *crp,
                char        *incuser,
                char        *remuser,
                char        *rcmd,
                int         *fd2p)
{
    int     pid;
    int     x[2];
    int     y[2];
    int     z[2];
    int     fd_w;
    int     fd_r;
    int     fd_err;
    int     fd_2;
    char    *client_home = NULL;
    char    *tp;
    struct  sigaction *vp;
    struct  sigaction   new_act;
    struct  sigaction   old_act;
    int     istat;

    /*
     * now open pipes that will become daemons stdin, stdout, stderr
     * remember that x[0] is for read and x[1] is for write
     */
    if ((-1 == pipe(x)) || (-1 == pipe(y)) || (-1 == pipe(z)))
    {
        WritePmtStringMsg(
            crp->ap_w_prog_fd, EBMERRORMSG_AUXPROC_ERROR, 0,
            "Unable to create a pipe\n");
        return NULL;
    }
    fd_w   = x[1];      /* pipe for child's stdin */
    fd_r   = y[0];      /* pipe for child's stdout */
    fd_err = z[0];      /* pipe for child's stderr */

    /*
     * Prepare sigaction parameters
     *   ignore SIGCHLD during this
     */
    sigemptyset(&new_act.sa_mask);
    new_act.sa_handler = SIG_IGN;
    new_act.sa_flags   = SA_RESTART;
    istat = sigaction(SIGCHLD, &new_act, &old_act);
```

```c
    if ((pid = EBR_FORK()) == 0)
    {
        /*
         * child processing goes here while parent is stopped (
         *                                              if vfork())
         *
         * first, setup stdio to work using our pipes
         */
        if ((close(0)<0) || (close(1)<0) || (close(2)<0)
            || (close(x[0])<0) || (close(y[1])<0) || (close(z[1])<0))
        {
            WritePmtStringMsg(
                crp->ap_w_prog_fd, EBMERRORMSG_AUXPROC_ERROR, 0,
                "unable to close stdio\n");
            _exit(-1);
        }
        if ((dup(x[0])<0) || (dup(y[1])<0) || (dup(z[1])<0))
        {
            WritePmtStringMsg(
                crp->ap_w_prog_fd, EBMERRORMSG_AUXPROC_ERROR, 0,
                "unable to dup pipe ends for forked
                    child\n");
            _exit(-1);
        }
        if ((close(x[1])<0) || (close(y[0])<0) || (close(z[0])<0))
        {
            WritePmtStringMsg(
                crp->ap_w_prog_fd, EBMERRORMSG_AUXPROC_ERROR, 0,
                "unable to close pipe ends for forked
                    child\n");
            _exit(-1);
        }

        /*
         * At this moment the process has a real UID of the user
         * executing [x]abrecover (
         *                          since [x]abrecover has
         * the setuid bit set and is owned by root )
         *
         * [x]abrecover and an effective UID of root (
         *                                             executing
         * The first problem is the access(
         *                              ) function call uses the real UID
         * of the process.  So if the user is not root or or examin,
         *                                                     the access
         * function will fail.  This causes the recovery to fail.
         *                                                     Thus
         *
         * The second problem is System V Unix does not pass the
         *                                                     effective
         * to processes doing an exec of a shell.
         *                              Instead the effective UID
         * is changed back to the real UID of the process.  However, BSD
         *
         * implementation does maintain the effective UID of the
         *                                                     process.
         *
         * On System V,
         *             neither the real or effective UID will be root during
         * execution of the shell script if the user executing
         *                                                     [x]abrecover is
         * a non-root user.  This process will not be able to perform
         * /bin/sh -c
         *             <rcodename>/bin/startrec because this processes real
         */
```

```
2551  2      * and effective UID (which are now the same) does not have
2552  2      * permissions to the directory -ebadnih.
2553  2      *
2554  2      * The recmcpio process will be resetting the username of the
      2      *                                                  initiating
2555  2      * process from the standard input and setting the real UID to
      2      *                                                         the
2556  2      * UID of that user so it will do no harm to set the real UID
      2      *                                                      to root
2557  2      * at this point.
      2      *   It will be changed back to the actual user as one
2558  2      * of the first things in recmcpio.
2559  2      *
2560  2      */
      2      (void)setuid(getuid());
2562  2
      2      /*
      2       * now chdir to the directory home location
2564  2       */
2565  2      if (NULL == (pw = getpwnam(
2566  2                   remuser)))     /* lookup client home dir */
      2      {
2569  3          p = "Can't get home directory";
2570  3          goto default_home;
2571  3      }
2572  2      else
      2      {
2574  3          if (
2575  3              client_home = pw->pw_dir) != NULL)  /* if there is a pointer */
2577  4              if (0 != chdir(client_home))  /* cd to directory */
2578  5                  p = "Can't chdir to home";
2579  5                  goto default_home;
2580  4              }
2581  4          }
2582  3          else
      3          {
2584  4  default_home:
2585  4              WriteErrStringMsg(cmp->ap_w_prog_fd,
      5                  EDMERPROGMSG_AUXPROC_WARNING, 0,
2586  5                  "%s",
2587  5  \*%s\* for user \*%s\*, defaulting to /usr/epoch/EB\n",
2588  4                  p);
2590  4                  cmp->ap_w_prog_fd, client_home, remuser);
2591  4              client_home = "/usr/epoch/EB/CLIENT_HOME";
2592  4          }
2593  3      }
2595  2
      2      /*
2596  2       * client_home now points to the home dir of the client
2597  2       *                                                software
      2       */
2599  2
2600  2      /*
      2       * Make sure that the home directory that we finally ended
      2       *                                                     up with
      2       * is really there.
      2       *      Yes.    I know.
      2       *        I may have already done this if I found a passwd
      2       *        entry for the client backup username and it had a home
      2       *                                                       directory.
2601  2       */
2602  2
2603  2
2604  2
```

```
2605  2      /*
2606  2       * However,
      2       *    I have to do it again just in case I came from another
      2       *    path.
2607  2       *    I did not really feel like trying to fix up the spagetti
2608  2       *    code.  I'm feeling a little lazy today...
      2       */
      2      if (0 != chdir(client_home))
2610  2      {
2611  2          WriteErrStringMsg(
2612  2              cmp->ap_w_prog_fd, EDMERPROGMSG_AUXPROC_ERROR, 0,
2613  2              "Server backup directory '%s' not found
      2              or not searchable\n", cmd);
      2              client_home);
2614  2      }
2615  2
2616  1      _exit(-1);
2618  1  }
2619  2  (void)execl("/bin/sh", "sh", "-c", cmd, 0);
2620  2  WriteErrStringMsg(
      2      cmp->ap_w_prog_fd, EDMERPROGMSG_AUXPROC_ERROR, 0,
      2      "Unable to exec /bin/sh sh -c %s\n", cmd);
2622  1  _exit(127);
2623  1  }
2624  1
2625  2  /*
2626  2   * Close the parent's copies of the child-ends of the pipes
2628  2   */
2629  1  close(x[0]);      /* pipe for child's stdin  */
2631  1  close(y[1]);      /* pipe for child's stdout */
      1  close(l[1]);      /* pipe for child's stderr */
2633  2
2634  2  /*
2635  2   * check for fork() failure
2637  2   */
2638  2  if (-1 == pid)
2640  2  {
2641  2      WriteErrStringMsg(
      2          cmp->ap_w_prog_fd, EDMERPROGMSG_AUXPROC_ERROR, 0,
2643  2          "fork() failed\n");
2646  2      close(x[1]);      /* pipe for child's stdin  */
2647  2      close(y[0]);      /* pipe for child's stdout */
2648  2      close(l[0]);      /* pipe for child's stderr */
2649  2      if (istat == 0)
2651  2      {
2652  2          sigaction(SIGCHLD, &old_act, NULL);
2653  2      }
2655  2      return NULL;
2656  2  }
2657  2
2659  2  /*
2660  2   * now fill in fd's to be returned
2661  2   */
2663  2  ebr_direct_rcmd_fds[0] = fd_w;
      2  ebr_direct_rcmd_fds[1] = fd_r;
2664  2  if (NULL != fdp)
      2  {
      2      *fdp = fd_err;
      2  }
```

```
2666  2
2667  2          if (istat == 0)
2668  2          {
2669  1              sigaction(SIGCHLD, &old_act, NULL);
2670  1          }
2671  1      return ebr_direct_rcmd_fds;
2672  1  }
          /* end of ebr_direct_rcmd() */
```

```
2674      /*
2675       * routine to search the clients_installed file for a clients
                                                                  connection
2676       * method.  Returns a pointer to the connection method string if
                                                                  successful,
2677       * NULL if the client entry could not be found.
2678       */
2679
2680      static char *
2681  rb_getmethod(register char *host,
2682                   uint_t *concurrency,
2683                   uint_t client_type)
2684  1   {
2685  1       FILE                *file_ptr;
2686  1       FILE                *lock_ptr;
2687  1       FPDB_HOLDER         *fpdb_holder;
2688  1       fpdb_inclient_ty    *fpdb_inclient;
2689  1                           installed_client;
2690  1       eperrno
2691  1       errno;
2692  1       static char         rb_cl_lastmethod[32] = "";
2693
2694  #ifdef PARAM_CHECK
2695  1       if (NULL == host)
2696  2       {
2697  2           rbe_internal_error(RBRECOVER_MKERR(
2698  2                                  EINVAL), null_arg, "host name");
2699  2           return NULL;
2700  1       }
2701  #endif /* PARAM_CHECK */

2702
2703  1       /*
2704  1        * get pathname to file
2705  1        */
2706  2       if (fpdb_inclient_init(FPDB_INIT_DEFAULT_PATH) != 0)
2707  2       {
2708  2           return NULL;
2709  2       }
2710
2711  1       if (fpdb_inclient_path == NULL) /* if file name needs to be
                                                  prepared */
2712  1       {
2713  1           return NULL;
2714  1       }
2715
2716  1       /*
2717  1        * request was not in cache -- go load it
2718  1        */
2719
2720  2       if (lock_ptr = fpdb_lock_file(fpdb_inclient_path,
                                            "restore scanning
                                            clients_installed file",
2721  2                                    1, NULL, 0) == NULL)
2722  2       {
2723  1           return NULL;
2724  2       }
2725  1       if ((file_ptr = fpdb_open_file(fpdb_inclient_path)) == NULL)
2726  2       {
2727  2           (void)rbe_user_error(
2728  2                   fpdb_errno, "Unable to open file \"%s\" for scanning",
2729  2                   fpdb_inclient_path);
2730  2           fpdb_unlock_file(lock_ptr);
2731  1           return NULL;
2732  2       }
```

```
while ((inclient_holder = ftdb_read_record(
                           file.ptr, &errnum)) != NULL)
{
    installed_client = (ftdb_inclient_ty *)inclient_holder->idvec;
    if (errnum == 0)
    {
        if (ebc_same_host(
            host, installed_client->hostname)) /* a match? */
        {
            if (concurrency != NULL)
            {
                if (0 == installed_client->concurrency)
                {
                    *concurrency = 32767;
                }
                else
                {
                    *concurrency = installed_client->concurrency;
                }
            }
            if (client_type != NULL)
            {
                *client_type = installed_client->platform;
            }
            strcpy(rb_cl_lastmethod, installed_client->smethod);
            (void)ftdb_close_file(file.ptr);
            ftdb_unlock_file(lock.ptr);
            ftdb_destroy_holder(inclient_holder);
            return rb_cl_lastmethod;    /* return allocated string */
        }
        ftdb_destroy_holder(inclient_holder);
    }
}

/*
 * if we get here -- did not find host in clients.installed file
 */

(void)ftdb_close_file(file.ptr);
ftdb_unlock_file(lock.ptr);
return NULL;
/* end of rb_getmethod() */
```

```
/*
 * Reads remote file descriptor for reads of stderr
 * of the remote cmd.  Read is done until nothing is left
 * then returns with 0 or without exit code.
 * The error/warning messages come first and a dealt with.
 * if the exit code of the remote cmd is returned.
 *
 * (Inp) int fd -- The file descriptor to wait on and read
 * (Out) int *exitp -- The Exit code.
 * (Inp) char *remotehostname -- The remote client name.
 * (Inp) bool_ty first_call -- Is this the first call ??
 * (I/O) enum input_states *state_ptr
 * (I/O) enum input_states *next_state_ptr
 * (I/O) bool_ty *skipping_leading_whitespace
 * (I/O) char *parsebuf -- the parse position of the messages.
 * (I/O) int *parsePos -- the position of the logging message.
 * (I/O) int *msgPos to the position of the logging message.
 *
 * None of the I/O vars need to be initialized for the first call.
 *
 * Returns: bool_ty,
 * False: The remote command has not exited.
 * True: The remote command has exited.
 */

static bool_ty
parse_remote_stderr_info2(int fd,
                          int *exitp,
                          int remote_fd,
                          char *remotehostname,
                          bool_ty first_call,
                          enum input_states *state_ptr,
                          enum input_states *next_state_ptr,
                          bool_ty *skipping_leading_whitespace,
                          char *parsebuf,
                          int *parsePos,
                          int *msgPos)
{
    enum input_states  previous_state;
    static bool_ty     cookie;
    char               buf[REMFD_MAX_COOKIE_LEN+1];
    int                protralled = 0;
    int                c          = 0;
    char               done       = 0;
    int                ret_status;
    char               temp_exit_status;
    #define MSGBUFFLEN 260
    static char        msgLogBUf[MSGBUFFLEN + 1];
    bool_ty            ret_status;
    static int         temp_exit_status = FALSE;
    int                write_prog       = 1;

    *exitp = 0; /* Initialize to success. */

    if (TRUE == first_call)
    {
        /* first time processing. */

        if (debugmode)
            rlog_log_stats(
                0,
                "AUXPROC: Processing stderr info from fd %d\n",
                remote_fd);

        *state_ptr = INSTATE_BEGIN;
        *next_state_ptr = INSTATE_SEARCH_PREFIX0;
        *parsePos = 0;
    }
```

```
2818  2          *exitp = 0;
2819  2          *segPos = 0;
2820  2          *exit_status = 0;
2841  2          *skipping_leading_whitespace = FALSE;
2842  2          n = 0;
2843  3      }
2844  3      else
2845  2      {
2845  2          n = *parsePos;
2846  2      }
2847  3
2849  1      while ((profailed && !done)
2850  2      {
2851  2          int start_ec;
2852  2          int c;
2855  2
2856  2          /*
2857  2           * Wait for the next character from the remote
2858  2           * stderr stream.  Ignore interrupts, unless
2859  2           * they were due to the attention signal.
2860  2           */
2861  2          start_ec = attn_ec;
2862  2          do
2863  3          {
2864  3              r = fd_avail_test_int(remote_fd);
2865  3              if (0 == r)
2866  4              {
2867  4                  return ret_status;
2868  4              }
2871  3          } while (r == -1 && errno == EINTR && attn_ec == start_ec);
2872  2
2873  2          /*
2874  2           * now that there is a character, read it
2875  2           */
2876  2          c = 0;
2877  2          do
2878  3          {
2879  3              r = CDL_read(remote_fd, &c, 1, 0);
2880  3          } while (r == -1 && (errno == EINTR &&
2881  3                  attn_ec == start_ec));
2882  2
2884  2          if (r != 1)
2885  2          {
2886  2              *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2887  3              profailed = 1;
2888  3              ret_status = TRUE;
2889  3              break;
2890  2          }
2892  2          previous_state = *state_ptr;
2893  2          *state_ptr = *next_state_ptr;
2895  2          switch (*state_ptr)
2896  2          {
2897  2          case INSTATE_SEARCH_PREFIX0:
2898  2              if (debugmode)
2899  4              {
2900  4                  rbe_log_state(0, "AUXPROC: {
2901  4                      %c) INSTATE_SEARCH_PREFIX0\n", c);
```

```
2903  4              if (c == RMMFD_MAGIC_PREFIX(0))
2904  4              {
2905  4                  *next_state_ptr = INSTATE_SEARCH_PREFIX1;
2906  4                  *parsePos = n;
2907  4              }
2908  4              break;
2909  3
2911  3          case INSTATE_SEARCH_PREFIX1:
2912  3              if (debugmode)
2913  4              {
2914  4                  rbe_log_state(0, "AUXPROC: {
2915  4                      %c) INSTATE_SEARCH_PREFIX1\n", c);
2917  3              }
2918  3              if (c == RMMFD_MAGIC_PREFIX(1))
2919  4              {
2920  4                  *next_state_ptr = INSTATE_GATHER_COOKIE;
2921  4                  n = 0;
2922  4                  *parsePos = 0;
2923  3              }
2924  3              else
2925  4              {
2926  4                  *next_state_ptr = INSTATE_SEARCH_PREFIX0;
2927  4                  n = 0;
2928  4                  *parsePos = 0;
2929  3              }
2931  3              break;
2932  3
2934  2          case INSTATE_GATHER_COOKIE:
2935  3              if (debugmode)
2936  4              {
2937  4                  rbe_log_state(0, "AUXPROC: {
2938  4                      %c) INSTATE_GATHER_COOKIE\n", c);
2940  3              }
2941  3              if (c == RMMFD_MAGIC_SUFFIX(0))
2942  4              {
2943  4                  /*
2944  4                   * Got the cookie
2945  4                   */
2947  4                  cookie[buf[n] = '\0';
2948  4                  *next_state_ptr = INSTATE_SEARCH_SUFFIX0;
2949  4                  n = 1;
2950  4                  *parsePos = 1;
2951  3              }
2952  3              else if (strchr(RMMFD_COOKIE_CHARS, c) == NULL)
2953  4              {
2954  4                  /*
2955  4                   * We found a valid prefix (else we would
2956  4                   * not be here) but the cookie contains an
2957  4                   * illegal character.  Should not happen;
2958  4                   * there has been some protocol failure.
2959  4                   */
2960  4                  *exitp = SPEXIT_REMOTE_STDERR_FAIL;
2961  4                  ret_status = TRUE;
2962  4                  profailed = 1;
2963  3              }
2964  4              else if (n == RMMFD_MAX_COOKIE_LEN)
2965  4              {
```

```
2966          /*
2967           *,
2968           *  Should have seen the SUFFIX[0] by now.
              *,
              */
2970          *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2971          ret_status = TRUE;
2972          profiled = 1;
2973      }
2974      else
2975      {
2977          /*
2978           * another cookie character.
              */
2980          cookidbuf[n] = c;
2981          n++;
2982          (*parsePos)++;
2983      }
2984      break;

      case INSTATE_SEARCH_SUFFIX:
          if (debugmode)
          {
              /*
               *,
               */
2990              the_log_stats(0, "AUXPROC: (
                       %c) INSTATE_SEARCH_SUFFIX\n", c);
          }

2992          if (c != REMFD_MAGIC_SUFFIX[n])
          {
2993              *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
2994              ret_status = TRUE;
2995              profiled = 1;
          }
          else
          {
2996              n++;
2997              (*parsePos)++;
2998              if (n == REMFD_MAGIC_LENGTH)
2999              {
3000                  *next_state_ptr = INSTATE_NEWLINE;
              }
          }
3001          break;

3006      case INSTATE_GATHER_STATUS:
3007          if (debugmode)
3009          {
3010              the_log_stats(0, "AUXPROC: (
3011                     %c) INSTATE_GATHER_STATUS\n", c);
3012          }
3013          if (isdigit(c))
3015          {
3016              char xx[2];
3017
3018              xx[0] = c;
3019              xx[1] = '\0';
3020
3022              temp_exit_status *= 10;
3023              temp_exit_status += atoi(xx);
3024              n++;
3025              (*parsePos)++;
3026              *skipping_leading_whitespace = FALSE;
3028          }
3029          else if (c == '\r')
```

```
3030          {
3031              *exitp = temp_exit_status;
3032              ret_status = TRUE;
3033              done = 1;
3034          }
3035          else if (c == ' ' &&
3036                 ((n == 0) || *skipping_leading_whitespace ))
3037          {
3038              /*
3040               * no-op, skip leading whitespace
3041               */
3042              *skipping_leading_whitespace = TRUE;
3043          }
3044          else
3045          {
3046              *exitp = SPEXIT_REMOTE_STDERR_PROTOCOL;
3047              ret_status = TRUE;
3048              profiled = 1;
3049          }
3052          break;

      case INSTATE_COPY_TO_STDOUT:
          /*
           * Eventually, the protocol should include
           * an explicit length for this state, so
           * we can do large read/writes and so we will not
           * be vulnerable to confusion based on gunk being
           * copied to stdout.
           *
           * For now, just shove the characters at stdout
           * and stop as soon as we fix PREFIX[0]
           */

3065          if (c != REMFD_MAGIC_PREFIX[0])
3066          {
              /*
               * build a buffer to write to the restore log
               */
3069              if ((c == '\n') == c) || ((*msgPos) >= MSGBUFFLEN))
3070              {
3071                  msgLogBuff[*msgPos] = '\0';
3072                  (void) rbe_log_stats(0,
3073                       "%s: %s",
3074                       remhostname,
3075                       msgLogBuff);
3076                  msgLogBuff[0] =
3077              }
3078              if (write_prog)
3079              {
3080                  WriteString(msg[prog_fd,
                            EXMMTERPROCMSG_RESTORE_RCMD_UNKNOWN,
3081                       0,
3082                       msgLogBuff);
3083              }
3085              else
              {
3086                  memset(msgLogBuff, 0, MSGBUFFLEN + 1);
3087                  *msgPos = 0;
3088              }
3090          }
3091          msgLogBuff[*msgPos] = c;
3093          (*msgPos)++;
```

```
3095  3                    }
3096  3                  else
3097  3                    {
3098  4                    *parsebuf = i;
3099  4                    *next_state_ptr = INSTATE_SEARCH_PREFIX;
3100  4                    break;
3101  4                    }
3102  3
          case INSTATE_NEWLINE:
            if (c == '\n')
              {
3104  3         *parsebuf = i;
3105  3         *next_state_ptr = INSTATE_SEARCH_PREFIX;
3106  4         break;
3107  5         }
3108  5
            if (previous_state == INSTATE_SEARCH_SUFFIX)
              {
3110  5         *next_state_ptr = decodecookie(cookiebuf);
                memset(cookiebuf, 0, REMFD_MAX_COOKIE_LEN+1);
3112  5         n = 0;
3113  4         *parsebuf = 0;
3114  5         }
3115  4       else
3116  5         {
3117  5         *next_state_ptr = INSTATE_SEARCH_PREFIX;
3118  4         }
3119  5       break;
3120  3       } /* end of switch */
3121  2     } /* end of while loop */
3122  1
3124  1   return ret_status;
3126  1
          /* end of parse_remote_stderr_info2() */
          }
```

```
3131
3132
3133      /*
           * ForwardXcpiogenProgress()
           * REP: Restore Engine Process. AP: auxproc. XC: xcpiogen.
           * This routine read progress messages from XC and forwards
           * them to REP. This routine should not block on a read().
           *
           * Args:
           *   int xcpiogen_prog_fd -- the progress channel from XC to AP.
           *   int restore_engine_prog_fd -- the progress channel from AP to REP.
           *
           * Returns:
           *   int -- the number of forwarded messages. -1 for an error.
           *
           * Side Effects:
           *   Read and writes file descriptors.
           */
          static int
          ForwardXcpiogenProgress(int xcpiogen_prog_fd,
                                  int restore_engine_prog_fd,
                                  boolean_t *zero_byte_read)
3153      {
3154  1     int bytes_skipped = 0;
3155  1     int read_ret, write_ret;
3156  1     int fd_test;
3157  1     int msg_count = 0;
3158  1     int prog_chan_msg = xcpiogen_prog_fd;
3159  1     int debug_prog = 0;
3160  1     int write_prog = 1;
3161  1
3162  1     *zero_byte_read = FALSE;
3164  1
3166  1     memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
3168  1
3170  1     while(1 == (fd_test = fd_avail_test(xcpiogen_prog_fd)))
3171  1       {
3172  2       read_ret = ReadMsg(
                      xcpiogen_prog_fd, &xcpiogen_msg, &bytes_skipped);
3174  3
3176  2       if(0 != bytes_skipped)
3177  3         {
3178  2         *zero_byte_read = TRUE;
3179  3         break;
3180  3         }
3181  3       else if(-1 == read_ret)
3182  3         {
3183  3         break;
3184  3         }
3185  3
3186  3       if(debug_prog)
3187  3         {
3188  2         (void) rhe_log_stats(0,
3189  3              "debug progress"
3190  3              DEBUG_PROG_MESSAGE_PEAK(&xcpiogen_msg));
                  }

              if(0 == read_ret)
                {
                (void) rhe_log_stats(0,
                     "The progress channel from xcpiogen had %d"
                     extraious bytes.");

              if(write_prog)
```

```
3191 2        write_ret =    WriteMsg(restore_engine_prog_fd, &xcpiogen_msg)/
3192 2        msg_count++;
3193 2        free(xcpiogen_msg.pcm_body');
3194 2
3196 1        memset(&xcpiogen_msg, 0, sizeof(prog_chan_msg));
3197 1
3198 1        if(-1 == fd_test)
3199 1        {
3200 0           (void) rbe_log_status(RBRECOVER_MKERR(errno),
3199 0                   "Encountered error testing xcpiogen file
3200 0                                     descriptor.");
3201 1
3202 1        return -1;
3203 1        }
3204 1
3205 2     return msg_count;
       }
```

```
3207   /*
3208    * static int DemuxAuxChildren()
3209    *
3210    * This function handles IRC communications between the following
3211    * processes:  REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
3212    *
3213    * and RC: remote command. The RC sends error and warning messages
3214    * to AP. When the RP is finished, RC's exit status is sent last.
3215    * The remote exit status indicates that the restore is finished.
3216    * RP error and warning messages are logged and forward on as
3217    * progress messages to RP. At the same time XC will send progress
3218    * information to AP, and AP will forward them to REP. XC does its
3219    * own logging.
3220    *
3221    * Args:
3222    *   int progress_fd -- (in) this file descriptor is from AP to REP.
3223    *   int remote_fd -- (in) this file descriptor is from RC to AP.
3224    *   char *remote_progname -- (in) this is the RP executable name.
3225    *   int xcpiogen_fd -- (in) this file descriptor is from XC to AP.
3226    *   int xcpiogen_pid -- (in) the pid for XC.
3227    *   int *remote_exit -- (out) the remote exit interpreted.
3228    *
3229    * REP: Restore Engine Process, AP: auxproc, XC: xcpiogen,
3230    *                                            RC: remote command.
3231    *
3232    * Returns: ??
3233    *
3234    */
3235   static int
3236   DemuxAuxChildren(int progress_fd,
3237                    int remote_fd,
3238                    char *remote_progname,
3239                    int xcpiogen_fd,
3240                    int xcpiogen_pid,
3241                    int *remote_exit)
3242   {
3243       int remote_exitinfo;
3244       int forward_ret;
3245       boolean_ly remote_exited = FALSE;
3246       struct pollfd monitor_fds[2];
3247       int poll_ret;
3248       boolean_ly remote_first_call = TRUE;
3249       int logging_channel = 0;
3250       int start_ac = start_ac_global;   /* attr_ac global */
3251       boolean_ly xcpiogen_zero_byte_reads;
3252       int number_fds;
3253       /* The below args are needed by and maintained by
3254        * parse_remote_stderr_info(). No need to initialize
3255        * them. They simply maintain state information
3256        * for multiple invocations of parse_remote_stderr_info(). */
3257       enum input_states remote_state_ptr;
3258       enum input_states remote_next_state_ptr;
3259       boolean_ly skip_whitespace;
3260       int parsepos;
3261       int RegPos;
3262
3263       number_fds = 2;
3264       monitor_fds[0].fd = remote_fd;
3265       monitor_fds[0].events = POLLIN;
3266       monitor_fds[0].revents = 0;
```

```
3270            monitor_fds[1].fd = xcpiogen_fd;
3271            monitor_fds[1].events = POLLIN;
3272            monitor_fds[1].revents = 0;
3273
3274        while(!(FALSE == remote_exited() &&
3275                poll_ret = CHLD_poll_read(
3276                        monitor_fds, number_fds, 5000)) ||
3277              (poll_ret > 0) ||
3278              ((EINTR == errno) && (-1 == poll_ret) &&
3279               start_ec != attn_ec)))
3280        {
3281            if (0 == poll_ret)
3282            {
3283                /* Timed out */
3284                /* check for the attention signal and other periodic
3285                 * checks. */
3286
3287            if (start_ec != attn_ec)
3288            {
3289
3290            /* Timeout on read, AND we were cancelled (SIGUSR1) --
3291             * If this is an SSL socket read timeout, the client may
3292             * have gone away without us knowing it,
3293             * so break out of the loop
3294             * without the remote exit status.
3295             *
3296             * Do this for all remote
3297             * process connections.  Since the xcpiogen to recxpio
3298             * connection might be SSL, the recxpio in this case
3299             * the remote_fd.  In this case
3300             * the remote process will not know that xcpiogen is
3301             * gone.
3302             */
3303
3304                remote_exit = SPEXIT_REMOTE_NO_STATUS;
3305                                        /* leaving w/o status */
3306                the_user_error              (0,
3307                    "No remote exit status received in 5
3308                     seconds -- stopped
3309                    "waiting for aborted restore's remote
3310                     process to finish");
3311
3312                break;
3313            }
3314
3315            else if (-1 == poll_ret)
3316            {
3317                /* EINTR */
3318                /* log this here */
3319                *the_log_stats(errno, "auxproc -- ");
3320            }
3321            else
3322            {
3323                /* Xcpiogen process is expected to send progress
3324                 * messages,
3325                 * including warnings and errors along with restore
3326                 * progress over the stderr channel.  We will check
3327                 * prior to the remote channel.  We presume that the
3328                 *                  This presumes that the remote
3329                 * channel determines whether we are done with restore.
3330                 */
```

```
3320                 */
3321                monitor_fds[1].revents = 0;
3322
3323                xcpiogen_zero_byte_reads = 0;
3324
3325                forward_ret = ForwardXcpiogenprogress(
3326                        xcpiogen_prog_fd,
3327                        progress_fd,
3328                        xcpiogen_zero_byte_reads);
3329            }
3330
3331            if(TRUE == xcpiogen_fds[1].revents)
3332            {
3333                /* Lets no longer wait on this fd,
3334                 *  and get a zero byte read.  The
3335                 *  second fd is for xcpiogen.
3336                 */
3337                number_fds = 1;
3338            }
3339
3340            /* POLLHUP/BAND Why do we need these conditions. */
3341
3342            /* POLLHANGUP & monitor_fds[0].revents
3343             * (POLLERRNORM & monitor_fds[0].revents ||
3344             *  POLLIN & monitor_fds[0].revents ||
3345             *  POLLHUP & monitor_fds[0].revents)
3346             */
3347            if( POLLERRNORM & monitor_fds[0].revents ||
3348                POLLERRBAND & monitor_fds[0].revents ||
3349                POLLIN & monitor_fds[0].revents ||
3350                POLLHUP & monitor_fds[0].revents)
3351            {
3352                /*
3353                 * Remote process is expected to send information back
3354                 * stream.  Read that information, parse it,
3355                 *                              and if available
3356                 * send remote exit status back to parent
3357                 *                        If the exit status
3358                 * is read this loop is terminated.
3359                 */
3360
3361                remote_exited = parse_remote_stderr_info2(progress_fd,
3362                        remote_fd,
3363                        &remote_state_ptr,
3364                        &remote_next_state_ptr,
3365                        &skip_whitespace,
3366                        &posraw0o,
3367                        &bkgless);
3368
3369                        &remote_exitinfo,
3370                        &remote_programe,
3371                        remote_first_call,
3372
3373                remote_first_call = FALSE;
3374            }
3375            }
3376
3377            /* Should we test xcpiogen for a last time before we return? */
3378
3379            if(1 == fd_avail_test(xcpiogen_prog_fd))
3380            {
3381                xcpiogen_zero_byte_reads = FALSE;
3382            }
3383
3384            forward_ret = ForwardXcpiogenprogress(xcpiogen_prog_fd,
```

```
3373  2
3374  2            }
3375  1                                  progress_fd,
                                         &xcplogen_zero_byte_reads);
3376  1            }
3377  1            if (-1 == poll_ret)
3378  1            {   /* Noc einre */
3379  2               rbe_log_stats(RBRECOVER_MKERR(errno),
3380  2                  "Auxproc failed to poll children pids!");
3381  2               return -1;
3382  1            }
3383  1            if(TRUE == remote_exitted)
3384  2            {
3385  2               *remote_exit = remote_exitinfo;
3386  2               return 0;
3387  1            }
3388  1         }
3389       } /* DemuxAuxChildren() */
```

D12

```c
/*
 *****************************************************
 **
 **  File Name:    RSLinitin.c
 **
 **  Copyright (c) 1998,1999 by EMC Corporation.
 **
 **  Purpose:
 **
 **      this module contains the Restore Service Library
 **                      functions
 **      Initialize and terminate the restore operation.
 **
 **
 **  Table of Contents:
 **
 **          RSTSL_Initialize
 **          RSTSL_Finish
 **
 **          Internal functions
 **
 **
 **  Compile-Time Options:
 **      This section must list any compile time definitions
 **      which will affect this header.
 *****************************************************/

/* The following provides an RCS id in the binary that can be located
 * with the what(1) utility. The intent is to keep this short.
 */

#ifndef LINT
static char RCS_id [] = "$RCSfile$ "
                        "$Revision$ "
                        "$Date$" ;
#endif


/*
 * Feature test switches.
 * Standard defines required to turn on OS features go here.
 *
 * The following is required for code that uses POSIX API's.
 * Remove for non-POSIX, non-portable code.
 */

#define _POSIX_SOURCE 1


/*
 * System headers.
 */
#include <sys/param.h>
#include <dirent.h>
#include <fcntl.h>


/*
 * Epoch headers.
 */
#include <eb/eb-port.h>
#include <eb/rb_log.h>
```

```c
/*
 * Local headers
 */
#include <RSLinterin.h>
#include <RSLstartup.h>


/*
 * #defines, structures, typedefs local to this source file
 */
static eerrno_by init_plugin( restore_context *rcp );
static int validate_plugin( struct plugindata *pdp );


/*
 * External declarations
 */
/*
 * This is the global "restore context" that will be used
 * throughout the rest of the restore operations.
 */
struct restore_context *rcp = NULL;


/*
 * Definitions of the names of the plugin functions in the pfuncArray
 * of the plugindata structure.
 *     These must be in the same order and position
 * as the pfuncIndex values defined in RSLplugin.h.
 */
char *pfFuncNames[PFuncIndexLast+1] = {
    "RSTPI_Identify",
    "RSTPI_Initialize",
    "RSTPI_GetPluginObjects",
    "RSTPI_SetPluginObject",
    "RSTPI_GetNextLevelObjects",
    "RSTPI_ClassRestoreContext",
    "RSTPI_Submit",
    "RSTPI_GetTopLevelTemplates",
    "RSTPI_DoesAlternateExist",
    "RSTPI_MaxObject",
    "RSTPI_ImmxObject",
    "RSTPI_IsObjectMarked",
    "RSTPI_IsObjectMarkable",
    "RSTPI_GetAllBackupTime",
    "RSTPI_GetCurrentBackupForTime",
    "RSTPI_SetBackupForTime",
    "RSTPI_SetPrevBackup",
    "RSTPI_SetNextBackup",
    "RSTPI_SetFirstBackup",
    "RSTPI_SetMostRecentBackup",
    "RSTPI_IsTherePrevBackup",
    "RSTPI_IsThereNextBackup",
    "RSTPI_IsThereNextBackupForTime",
    "RSTPI_Finish",
    "RSTPI_StartRestore",
    "RSTPI_IsRestorableObjects",
    "RSTPI_GetPortableObjects",
    "RSTPI_GetFindResults",
    "RSTPI_GetNecessaryMedia"
};
```

```c
127
128
129    /**********************************************************
130  1  * RSTSL_Initialize()
131  1  *
132  1  * This function takes care of all the initialization for a restore
133  1  * session. This must be called prior to any of the other functions
134  1  * in the Restore API.
135  1  *
136  1  * Parameters:
137  1  *
138  1  * userName (I) - The name of the user.
139        *
140        **********************************************************/
141  1  errno.ty
142  1  RSTSL_Initialize( const char *userName )
143  1  {
144  1      errno.ty status = R_SUCCESS;
145
146          /*
147           * If we have not yet allocated space for a restore context
148  1        * structure, do so now. If we have already done so, just clear it
149  1        * now.
150  1        */
151  2
152  2      if (NULL == rcp)
153  2      {
154  2          rcp = (struct restore_context *)malloc(sizeof(
155  2                                struct restore_context));
156  2
157  2          if (NULL == rcp)
158  2          {
159              rec_spl_log_csm(SUB_CSM_NOMEM, NULL);
160              return(EP_RB_RECOVER_NOMEM);
161          }
162      }
163      memset(rcp, 0, sizeof(struct restore_context));
164
165  1      rcp->rtc_human_uidname = esl_strdup( userName );
166  1
167  1      if (!(rcp->rtc_human_uidname = esl_strdup( userName )) {
168  1          rec_spl_log_csm(SUB_CSM_NOMEM, NULL);
169          return(EP_RB_RECOVER_NOMEM);
170  1      }
171  1
172  1      /*
173  1       * Set the appropriate field in the recovery context to indicate
174  1       * that this recovery session is based on the Recover API.
175  1       * This flag is in place for historical reasons but is used by
176           * other parts of the Recover API library.
177           */
178      rcp->gui_mode = 1;
179  1
180  1      /*
181  1       * Initialize the logging mechanism.
182           */
183  1      if (status = rtrlog_begin(rcp, progname))
184  2          return(status);
185  2
186  1      /*
187           * Initialize the few "recover context" variables that we can at
188           * this early stage.
```

```c
191        */
192      setup_proc(rcp);
193
194        /*
195  1     * The following call will:
196  1     * - initialize the savesel database.
197  1     * - find any information we can at this point.
198  1     * - Inter any information we can at this point.
199        */
200
201      if (status = startup(rcp))
202  2      return(status);
203  2
204  1      /* Do plugins setup: Find and initialize all valid restore plugin
205               libs. */
206  1      status = init_plugins( rcp );
207
208      return( status );
209          /* End of RSTSL_Initialize() */
210  1  }
211
```

```c
/*
***************************************************
* RSTSL_Finish
*
* Function Description:
*
* This function terminates a restoral session.
  but not while a restore is in
* progress.   It will
  be rejected if a restore is currently being executed.
* This routine will clean up any local memory used in the session.
*
* Parameters:
*
* none
*
*
***************************************************
*/

errno_ty
RSTSL_Finish( void )
{
    int rc_n;

    errno_ty err = E_SUCCESS;

    if ( NULL == rcp )
        return( E_SUCCESS );

    RemoveSubmitFiles();

    /*
    * Call rbc_cleanup() which kills the aux proc(
    * s), unlocks the work
    * item, then calls the rbc_log_end(
    * ) to enter the last logs and to close
    * the log file.
    */
    rbc_cleanup(rcp);

    /*
    * Deallocate the memory of restore_context and the related
    * structures.
    */
    mcst_destroy(rcp->rc_mcp);     /* Free the multicast structures */

    if (NULL != rcp->rc_mcp)
        free(rcp->rc_mcp);

    /*
    * Free the mark bit map space
    */
    for (mc_n = 0; mc_n < rcp->rc_marks_plane_alloc; mc_n++)
    {
        if (NULL != rcp->rc_marks[mc_n])
            free(rcp->rc_marks[mc_n]);
    }

    rcp->rc_marks[mc_n] = NULL;

    if (NULL != rcp->rc_marks_by_plane)
```

```c
    {
        free(rcp->rc_marks_by_plane);
    }

    /*
    * Free the configuration structures
    */
    if (NULL != rcp->rc_cfgname)
    {
        free(rcp->rc_cfgname);
    }
#endif

#if 0
    if (NULL != rcp->rc_config)
        rbc_freeconfig(rcp->rc_config);
#endif

    /*
    * Free the DS_NOWB structures array
    * Note that even though rc_dsnones is the head of linked list
    * of dsnone_info structure, the list is allocated via malloc
    * as an array initially (ref. alloc_plane_array()), therefore
    * we can do a free here.
    */
    if (NULL != rcp->rc_dsnones)
        free(rcp->rc_dsnones);

    /*
    * Free the volume list structures.
    */
    if (NULL != rcp->rc_ebvlist)
        (void)ebvl_voidlist_destructor(
            rcp->rc_ebvlist, EBVL_DESTROY_ALL);

    /*
    * Free the plugin related data
    */
    rcp->rc_backup_app = 0;
    while (rcp->currentPgptr = rcp->rc_ppilist)
    {
        rcp->rc_backup_app++;
        rcp->rc_ppilist = rcp->currentPgptr->next;
        /* allow plugin to clean up and close so: */
        if ( E_SUCCESS != (err =
            rcp->currentPgptr->plFuncArray[ plFuncIndexFinish ] (
                rcp)))
        {
            /* log error, continue */
            the_user_error( err,
                "RSTSL_Finish failed for restore plug-in",
                "library %s",
                (struct pluginInData *)
                rcp->currentPgptr->libHdl )-> name );

            dlclose( rcp->currentPgptr->libHdl );
            rcp->rc_ppilist = rcp->currentPgptr->next;
        }
```

```
336  2            free (rcp->current?iptr);
337  1        }
339  1
340  1        /*
341  1         * Free the various simple string buffers
     1         */
343  1        if (NULL != rcp->rc_top_level_object_name)
344  2            free (rcp->rc_top_level_object_name);
345  2
346  1
348  1        if (NULL != rcp->rc_template_name)
349  2            free (rcp->rc_template_name);
351  2
352  1
353  1        if (NULL != rcp->rc_workitem_name)
354  2            free (rcp->rc_workitem_name);
356  2
357  1
358  1        if (NULL != rcp->rc_human_uidname)
359  2            free (rcp->rc_human_uidname);
360  2
361  1
363  1        if (NULL != rcp->rc_effective_uidname)
364  2            free (rcp->rc_effective_uidname);
365  2
366  1
     1        /* don't free, its internal: free(rcp->rc_effective_uidname);
     1        */
369  1
370  2        if (NULL != rcp->rc_client_rbuname)
371  1            free(rcp->rc_client_rbuname);
373  1
374  2        if (NULL != rcp->rc_client_hostname)
375  2            free(rcp->rc_client_hostname);
376  2
378  1        if (NULL != rcp->rc_client_scriptname)
379  2            free(rcp->rc_client_scriptname);
380  2
     1        /* don't free, its internal: free(rcp->rc_client_scriptname);
     1        */
383  1        if (NULL != rcp->rc_client_dirtop)
384  2            free(rcp->rc_client_dirtop);
385  2
386  2
388  1        if (NULL != rcp->rc_cmd_context)
389  2
390  2            /* don't free -- its internal/temp data: free(
391  1                rcp->rc_cmd_context); */
393  1        if (NULL != rcp->rc_source_client_hostname)
394  2            free(rcp->rc_source_client_hostname);
395  2
396  2
398  1        if (NULL != rcp->rc_cpiogen_executable)
     1            free (rcp->rc_cpiogen_executable);
```

```
399  2        {
400  2            /* don't free, its internal: free(rcp->rc_cpiogen_executable);
     1            */
401  1
403  1        if (NULL != rcp->rc_plugin_wl_types)
404  2            free (rcp->rc_plugin_wl_types);
405  2
406  2
408  1        if (NULL != rcp->rc_pwd)
409  2            free(rcp->rc_pwd);
410  2
411  1        }
413  1        /*
414  1         * Finally, deallocate the restore_context itself
415  1         */
417  1        free (rcp);
418  1        rcp = NULL;
     1
422  1        return( err );
423  1        }    /* RSTSL_finish */
```

```
427
428
429   /************************************************************
430    *
431    *  Function Description:
432    *
433    *     init_plugins
434    *
435    *  This function locates, opens, validates and initializes all restore
436    *  plug-in (shared) libraries.  They must be located in
437    *  /usr/apcch/SES/cure_plugin
438    *        eb_cure_plugin(_dir)  ... so files in that
439    *  directory are opened and validated for version and presence of all
440    *  mandatory functions.
441    *
442    *  library to determine which optional features are supported and that
443    *  the corresponding functions are present.  Finally, the RSTPl_Initialize
444    *
445    *  function is called for each valid library.
446    *
447    *        The RSTPl_Identify function is called for each
448    *  Parameters:
449    *
450    *    Inputs:
451    *      rcp          (I)    - Pointer to restore context
452    *
453    *    Outputs:
454    *      none
455    *
456    *  Returns:
457    *      E_SUCCESS or EP_RM_RECOVER_XXX
458    *
459    *  Logic/pseudo code:
460    *
461    *    open plugin dir
462    *    while read_next_entry succeeds
463    *      verify .so file
464    *      open shared library (else continue)
465    *      on errors below:
466    *        close shared library file
467    *        continue
468    *      fetch all mandatory function addresses
469    *      validate version number
470    *      fetch Identify function
471    *      fetch all indicated optional function addrs
472    *      validate function
473    *      add validated types to composite exclusion list
474    *      add to valid plugin list
475    *    close plugin dir
476    *
477    *
478    ************************************************************/
479
480   static errno_t init_plugins( restore_context *rcp )
481   {
482       DIR                    *dirp;
483       struct dirent          *direntp;
484       errno_t                 status = E_SUCCESS;
485       struct pluginData      *pListePtr = NULL;
486       struct pluginData      *pListePtr = NULL;
487       int                     val_result;
488       struct pluginInitData  *idDataPtr;
489       char                   *tmp_typeso;
490       int                     shlib_dirlen;
491       char                    shlib_path [MAXPATHLEN];
```

```
484       /* open plugin directory or bust */
485       if ( NULL == (dirp = opendir( eb_cure_plugin_dir )) )
486       {
487           rec_api_log_cmt( SUB_CRM_PLUGIN_ERR, NULL );
488
489           return E_SUCCESS;    /* allow continuation w/o plugins */
490       }
491
492       return EP_RM_RECOVER_NO_PLUGINS;    /* later do this */
493   }
      #endif
      #else
      #if 1
      #endif

505       strcpy( shlib_path, eb_cure_plugin_dir );
506       strcat( shlib_path, "/" );
507       shlib_dirlen = strlen (shlib_path);
508
509       /* loop thru entries in directory */
510       while (NULL != (direntp = readdir( dirp )))
511       {
512           if (NULL == strstr( direntp->d_name , ".so" ) )
513               continue;    /* skip this guy */
514
515           strcpy( &shlib_path[shlib_dirlen], direntp->d_name );
516           if (NULL ==
517               (pListePtr =
518               = dlopen( shlib_path, RTLD_NOW )))
519           {
520               /* allocate next plugin data structure */
521               if (NULL ==
522                   (pListePtr =
523                   = called l, sizeof(
524                   struct pluginData) )))
525               {
526                   status = EP_RM_RECOVER_NOMEM;
527                   break;    /* fail thru to cleanup */
528               }
529
530               the_user_error( 0,
531               "Error opening restore plug-in library %s:\n",
532               %s: %s\n",
533               direntp->d_name, dlerror() );
534               continue;    /* skip this one */
535           }
536
537           if (0 != (val_result = validate_plugin(
538               pListePtr ) ) )
539           {
540               if (val_result == -1 || val_result == -4)
541               {
542                   the_user_error( 0,
543                   "Functions missing from restore plug-in library %s:\n",
544                   direntp->d_name, dlerror() );
545               }
546               else if ( val_result < 0 )
547               {
548                   the_user_error( 0,
549                   "Validation failed for restore plug-in
550                   library %s\n",
551                   direntp->d_name );
```

```c
542 3            else
543 4            {
544 4                rbe_user_error( val_result,
545 4        "RSTPI_identify failed for restore plug-in library
                                                        %s\n',
546 4                direntp->d_name );
547 3            }

549 3            diclose( plDataPtr->libhdl );
550 3            plDataPtr->libhdl = NULL;
551 3            continue;    /* on any error, close .so on errors */
552 3            }

        /* let DC plug-in do its initialization */
554 2            status =
555 2                rbe_user_error( plDataPtr->piFuncArray[piFuncIndexInitialize]( rcp );
555 2            if (E_SUCCESS != status)
557 1            rcp->appData = NULL;    /* enter plugin with clean appdata */

558 3                rbe_user_error( status,
559 3        "RSTPI_initialize failed for restore plug-in library
560 3                    %s\n',
561 3                dirntp->d_name );
562 2            diclose( plDataPtr->libhdl );    /* close .so on errors */
563 3            plDataPtr->libhdl = NULL;
564 3            status = E_SUCCESS;
565 3            continue;    /* this won't fatal */
566 2            }    /* on any error, skip this lib */

        /* add plDataPtr to valid plugin list */
570 2            if (NULL == plistPtr)
571 2                rcp->plistPtr = plDataPtr;
572 3            else
574 3                plistPtr->next = plDataPtr;    /* first in list */
575 2                                               /* link from prev */
576 2            plDataPtr = plDataPtr;
577 2            plistPtr = plDataPtr;              /* new end of list */

        /* save plugin's appdata */
580 2            plDataPtr->appData = rcp->appData;
581 2            rcp->appData = NULL;

        /* add workitem types to composite exclusion list */
585 3            idDataPtr = (struct pluginIdData *)plistPtr->idData;
586 2            if (idDataPtr->num_types > 0)
587 4            {
588 4                status = EF_RB_RECOVER_NOMEM;
589 4                break;
590 4            }

592 4            tmp_types = calloc( 1, 1 + idDataPtr->num_types
593 4                    + rcp->rc_num_plugin_wi_types );
        /* move old list to new buffer and free old list */
        memcpy( tmp_types,
                rcp->rc_plugin_wi_types,
                rcp->rc_num_plugin_wi_types );
```

```c
595 4                free( rcp->rc_plugin_wi_types );
596 3            memcpy( tmp_types + rcp->rc_num_plugin_wi_types,
597 3                    idDataPtr->wi_types,
598 3                    idDataPtr->num_types );
599 3            rcp->rc_num_plugin_wi_types +=
600 3                    idDataPtr->num_types;
601 3            tmp_types[rcp->rc_num_plugin_wi_types] = 0;
602 3            rcp->rc_plugin_wi_types = tmp_types;
603 3            }
604 2            }

606 1            (void)closedir( dirp );

608 1            /* free up leftovers */
609 1            if (NULL != plDataPtr)
610 1                free (plDataPtr);

612 2            if (E_SUCCESS != status)
613 2            {  /* free contents of plugin list */
614 1                while (NULL != plistPtr) {
615 1            /* allow plugin to clean up and close .so. */
616 2                plDataPtr->appData = plistPtr;
617 1                plDataPtr->piFuncArray[piFuncIndexFinish]( rcp );
618 2                diclose( plDataPtr->libhdl );
619 2                plistPtr = plDataPtr->next;
620 2                free (plDataPtr);
621 2            }
622 1            }

624 1            return status;
625 1        }
```

```
/* init_plugin */

/******************************************************************
 *
 * validate_plugin
 *
 * Function Description:
 *
 * This function retrieves the addresses of the mandatory plugin
 * functions
 *
 * and stores them in the function pointer array.  If any function is missing
 *
 * it returns -1.  It then calls the identify function and verifies plugin
 *
 * version, and finds its optional functions.  Specific error values are
 * returned on version mismatch and missing optional functions.
 *
 * Parameters:
 *
 * Inputs:
 *   piDataPtr
 *     I)  - Pointer to plugin data structure with libHdl set
 *
 * Outputs:
 *   piFuncArray is loaded with pointers to plugin functions
 *
 * Returns:
 *   0 on success
 *   -1 on any missing required functions
 *   -2 on version fails OR identify returns junk
 *   -3 if version type validation fails
 *   -4 on any missing optional functions indicated by options
 *      flags
 *
 *       EB_RB_RECOVER_xxx) for error codes returned from identify function
 *
 *****/

static int validate_plugin( struct pluginData *piDataPtr )
{
    int                 index;
    extern_by           status;
    struct pluginData  *piDataPtr;

    /* call identify and validate. */
    status = piDataPtr->piFuncArray[PIFuncIndexIdentify](
                    &piDataPtr->idData );

    if (status != R_SUCCESS)
        return status;

    if (NULL == (idDataPtr = (
                    struct pluginData *)piDataPtr->idData )

        return -1;

        for( index = 0; index <= PIFuncIndexLastBasic; index++ )
        {
            if (NULL == (piDataPtr->piFuncArray[index]
                    = (piDataPtr->libHdl,
                       piFuncNames[index] ))
            {

                return -1;

            }
        }
```

```
        if (idDataPtr->version != RSTPL_VERSION)
        {  /* only version 1 is supported so far */
            piDataPtr->idData = NULL;
            return -2;
        }

        if (idDataPtr->num_types && !(idDataPtr->wi_types)
        {  /* count cant be positive with null pointer */
            piDataPtr->idData = NULL;
            return -3;
        }
    }

    /* if startRestore option set, get its addr or bust */
    if ( ( piDataPtr->version & RSTPL_OPTION_SPECIAL_START)
         == (idDataPtr->options & RSTPL_OPTION_MASK_FIND))
         && ( NULL == (
              ( piDataPtr->piFuncArray[PIFuncIndexStartRestore]
                = (piDataPtr->libHdl,
                   piFuncNames[PIFuncIndexStartRestore] ))))
        return -4;

    /* OR if special find option set, get its addr or bust */
    if ( ( piDataPtr->options & RSTPL_OPTION_SPECIAL_FIND)
         == (idDataPtr->options & RSTPL_OPTION_MASK_FIND)
         && ( NULL == (
              ( piDataPtr->piFuncArray[PIFuncIndexFind]
                = (piDataPtr->libHdl,
                   piFuncNames[PIFuncIndexFind] )))))
        piDataPtr->idData = NULL;

    /* OR if special getMedia option set, get its addr or bust */
    if ( ( piDataPtr->options & RSTPL_OPTION_MASK_GET_MEDIA)
         == (idDataPtr->options & RSTPL_OPTION_SPECIAL_GET_MEDIA)
         && ( NULL == (
              ( piDataPtr->piFuncArray[PIFuncIndexGetMedia]
                = (piDataPtr->libHdl,
                   piFuncNames[PIFuncIndexGetMedia] )))))
        return -4;

    piDataPtr->idData = NULL;

    return 0;
}
```

723   /*
validate_plugin */

```c
/**********************************************************
**
** File Name:    RSLplugin.h
**
** Copyright (c) 1999 by BMC Corporation.
**
** Purpose:
**      Two-fold:
**          - Define prototypes of common functions exported from the
**            library
**          - Define prototypes of functions in a restore plugin library
**
**      These functions are part of the restore_legacy library.
**
**      This header defines the API that a backup application must supply to
**      be compatible with the new Restore API.  The Restore API provides
**      client-server based restore functionality formerly provided by the
**      RECOVER_API.  The plug-in API is provided in a shared library, designed
**      to be called by the Restore Service library -- the generic server side
**      component of the Restore API that executes as part of the Restore
**      Engine (server).
**
**
**  Compile-Time Options:
**      Need:
**          #define _POSIX_SOURCE 1
**
**      Need header files:
**          <ebutil/ebutil.h>
**          <restore/restore_api.h>
**          <restore/restoreRPC.h>
**
*****/

#ifndef H_RSLPIAPI
#define H_RSLPIAPI

#include <ebutil/ebutil.h>
#include <errno.h>
#include <restore/restore_api.h>
#include <restore/restoreRPC.h>

#ifdef __cplusplus
extern "C" {
#endif

/* The following definition identifies the version of the Restore Plugin
 * header that a plug-in library was built with.  Whenever a plugin function
 * prototype changes, or a plugin function is added, this value should be
 * updated.  It will be used by the restore engine code to verify
 * compatibility with individual restore plugin libraries.
 */
```

```c
#define RSTPI_VERSION    1

/* Definitions of the indices of the plugin function addrs in the
 * pifuncArray of the plugin function.
 */

/* Mandatory functions: */
#define PIFuncIndexIdentify            0
#define PIFuncIndexInitialize          1
#define PIFuncIndexGetUI               2
#define PIFuncIndexSetUI               3
#define PIFuncIndexGetIO               4
#define PIFuncIndexSetIO               5
#define PIFuncIndexClearRC             6
#define PIFuncIndexSubmit              7
#define PIFuncIndexTerminate           8
#define PIFuncIndexDoesAltExist        9
#define PIFuncIndexMark                10
#define PIFuncIndexUnmark              11
#define PIFuncIndexIsMarked            12
#define PIFuncIndexHandle              13
#define PIFuncIndexGetAllItems         14
#define PIFuncIndexCurTime             15
#define PIFuncIndexBkupTime            16
#define PIFuncIndexPrevTime            17
#define PIFuncIndexSetExtTime          18
#define PIFuncIndexGetMostRecent       19

/* Optional functions: */
#define PIFuncIndexPreview             20
#define PIFuncIndexPrevBkup            21
#define PIFuncIndexNextForTime         22
#define PIFuncIndexIsPrevForTime       23
#define PIFuncIndexFinish              24
#define PIFuncIndexGetResults          25
#define PIFuncIndexGetProgress         26
#define PIFuncIndexGetMedia            27
#define PIFuncIndexCurMedia            28
#define PIFuncIndexLast                28

/* Definitions of the names of the plugin functions in the pifuncArray
 * structure.
 *      These must be in the same order and position
 *      as the PIFuncIndex values above, and are initialized in RSLplugin.c
 */

char *pifuncNames[PIFuncIndexLast+1];

/* Definitions of the plugin/data structure. */

typedef eerrno_ty (*piFuncPtr)();       /* generic plugin func prototype */

/* callback function prototypes */

typedef struct restore_context restore_context;   /* for function prototypes */

/* restore plugin prototypes */

typedef boolean_ty  (*RSTPI_MarkProgressProc)(
                        unsigned long totalMarks;
    (*RSTPI_SubmitProgressProc)(
                        unsigned long totalElements);
```

```c
112 1   typedef boolean_t (*RSTPl_FindProgressProc) (
113 1                       unsigned long progressCount);
114 1   };
115
116 1   /** Definitions of the data that the RSTPl_Identify returns. **/
117 2
118 2   typedef boolean_t (*RSTPl_QuitSell)( void );
119
120 2   struct pluginData {
121 2
122 2       u_long version;      /* Version of the plugin header that the
123 2                               library was built with */
124 2
125 2       char *name;          /* Name of the backup application (64 byte
126 2                               buffer address) */
127 2
128 2                            /* Bit mask identifying the optional plug-in
129 2                               features supported. The bit definitions for
130 1                               this parameter are defined below. */
131
132 2       u_long options;
133
134 2       char *wi_types;      /* pointer to array of worktime types supported
135 2                               by the plugin */
136
137 2       u_short num_types;   /* Number of wi_types in the wi_types array */
138
139 1   };
140
141 1   /* The following bits in the RSTPl_Identify function
142 1    *                                             the find
143 1    * function for searching its catalog for specific objects
144 1    * Indicates if find is supported. The first bit
145 1    * supplies its own find function, the
146 1    * searches the most data can be used.
147 1    * second bit indicates whether the plugin
148 1    */
149
150 1   #define RSTPl_OPTION_MASK_START        0x1
151 1   #define RSTPl_OPTION_SPECIAL_START     0x1
152 1   #define RSTPl_OPTION_GENERIC_START     0x0
153
154 1   /* This bit indicates whether the plug-in supplies its own function to
155 1    * execute the actual restore. */
156
157 1   /* NOTE: The output structure containing this data is defined in
158 1    * RSLcontext.h */
159
160 1   #define RSTPl_OPTION_MASK_FIND          0x6
     1   #define RSTPl_OPTION_FIND_SUPPORTED     0x6
     1   #define RSTPl_OPTION_SPECIAL_FIND       0x4
     1   #define RSTPl_OPTION_GENERIC_FIND       0x2
     1   #define RSTPl_OPTION_NO_FIND            0x0

     1   /* The following flag indicates whether objects can be restored from
     1    * multiple backup planes in the same restore. If not, the user will be
     1    * warned that changing backup planes will cause objects marked for
     1    * restore from a different backup time to be automatically unmarked. */

     1   #define RSTPl_OPTION_MULTI_PLANE            0x8
     1   #define RSTPl_OPTION_MULTI_PLANE_SUPPORTED  0x8
```

```c
161      /* The following flag indicates whether the plugin provides an
162       *  RSTPl_GetNecessaryMedia function to return the list of media needed
163       *  to restore the restorable objects currently marked.
164       *  Otherwise the plugin must
165       *  maintain the ebvl_vollidlist_t * list in the restore_context
166       *  whenever objects are marked and unmarked.
167       */

168      #define RSTPl_OPTION_MASK_GET_MEDIA       0x10
169      #define RSTPl_OPTION_SPECIAL_GET_MEDIA    0x10
170      #define RSTPl_OPTION_GENERIC_GET_MEDIA    0x00

171      /* The following flag indicates whether objects can be restored */
172
173      #define RSTPl_OPTION_MASK_SYM_RESTORE     0x20
174      #define RSTPl_OPTION_SYM_RESTORE_ALLOWED  0x20
175
176      /***********************************************************
177       *  plug-in API function definitions
178       ***********************************************************/
179
180      /***********************************************************
181       * Identify: ('SCSI')
182       *
183       * a symmetric ('SCSI')
184       *
185       * This function is called once,
186       *              to identify and validate the plug-in
187       * regard to the operating restore engine.
188       *              the version number is checked
189       * for compatibility with the restore engine.
190       *
191       * of the plugin are specified.
192       *
193       * Parameters:
194       *   pi_defs - (
195       *      0) - address of the structure identifying the plugin to the
196       *           restore engine (
197       *           64 byte buffer address)
198       *   version - Version of the plugin header that the library was
199       *             built with
200       *   name    - Name of the backup application (64 byte buffer address)
201       *   options - Bit mask identifying the optional plug-in features
202       *             supported
203       *             The bit definitions for this parameter are
204       *             defined above.
205       *   wi_types - pointer to array of worktime types supported by the
206       *              plugin
207       *   num_types - number of wi_types in the wi_types array
208       *
209       * Returns:
210       *   E_SUCCESS         on success
211       *   EP_RS_RECOVER_xxx on error
212       ***********************************************************/

         /***********************************************************
          * Initialize:
          ***********************************************************/

         extern_ty RSTPl_Identify( const struct pluginData *pi_defs );
```

```
213   *
214 1 *  This function is called once, to allow the plug-in to perform its
215 1 *  'local' initialization.
216 1 *
217 1 *           The restore context will have already been
218 1 *  initialised with the config file parsed, and the human user fields
219 1 *
220 1 *  set:
221 1 *
222 1 *  Parameters:
223 1 *
224 1 *     context (I)  -  Pointer to the restore context
225 1 *
226 1 *  Returns:
      *     RL_SUCCESS         on success
      *     RL_RB_RECOVER_xxx  on error
      */
229   eerrio.ty RSTPl_Initialize( restore_context *context );

      /********************************************
231   *  Finish:
232   *
233 1 *  This function is called to allow the plug-in to clean up its
234 1 *
235 1 *  storage when a restore session is ending.
236 1 *
237 1 *  Parameters:
238 1 *     context (I)  -  Pointer to the restore context
239 1 *
240 1 *  Returns:
241 1 *     RL_SUCCESS         on success
242 1 *     RL_RB_RECOVER_xxx  on error
243 1 */
      eerrio.ty RSTPl_Finish( restore_context *context );

      /********************************************
248   *  Get Top Level Objects
249   *
250 1 *  This function is called to retrieve the configurable backup objects (work
251 1 *  items for network backups and work item sets for Symmetric Connect backups,
252 1 *
253 1 *  which are restorable for the given client.
254 1 *
255 1 *  It is a GOAL of this routine to return all objects ever backed
256 1 *  up successfully.  For network backups, though,
                               it only looks in the config
257 1 *  file for the 'top level objects' of the given client.
258 1 *
259 1 *  While the restore API will be called repeatedly to retrieve a  maximum
260 1 *  number of items on each call,
                               the plug-in cell must retrieve the whole
261 1 *
      *  set of applicable backup objects.
                     The plug-in cell restore service library
      *
      *  will manage the composite list of top level objects from all
                                                    backup apps.
      *
      *  Parameters:
262 1 *     context
263 1 *     sourceHost
264 1 *              (I)  -  the name of the host whose backups are being restored
265 1 *     topLevelObjs
266 1 *              (O)  -  ptr to linked list of Top Level Objects
```

```
267 1 *     numberEntries  (O)  -  the real number of objects returned in the list
      1 *
      1 *  Returns:
      1 *     RL_SUCCESS         on success
      1 *     RL_RB_RECOVER_xxx  on error
      1 */
      eerrio.ty
      RSTPl_GetTopLevelObjects( restore_context *context,
                                const char *sourceHost,
                                struct RSTPC_tlo_list **topLevelObjs,
                                short *numberEntries );

      /********************************************
      *  Set Toplevel Object
      *
287 1 *  This function is called to notify the plug-in that one of its top
      1 *  level
288 1 *  objects has been selected (for browsing and marking). The plug-in
      1 *  should perform whatever validation and initialization is needed to
      1 *  prepare
289 1 *
290 1 *  for browsing and marking this top level object.
291 1 *  If this call returns success:
292 1 *
293 1 *  called to retrieve the highest level restorable objects for this
294 1 *  backup object.
295 1 *
296 1 *  NOTE: This function is responsible for setting the template_name
      1 *  and
297 1 *  saveset_thread elements of the restore context.
298 1 *
299 1 *  Returns:
300 1 *     RL_SUCCESS         on success
301 1 *     RL_RB_RECOVER_xxx  on error
302 1 */
303   eerrio.ty
304 1 RSTPl_SetTopLevelObject( restore_context *context,
305 1                          struct RSTPC_top_level_obj
306 1                                              *backupObject );

      /********************************************
309   *  Get Next Level Objects.
310   *
311 1 *  This function is intended to allow retrieval of the children
312 1 *  of a given parent object.
313 1 *           The caller specifies the parent object and
314 1 *  whether or not to include bad files.  Even though the objects are
315 1 *  returned in a linked list, there could conceivably be thousands of
316 1 *  child objects, so the caller must specify the maximum number
317 1 *  of children to return.  The caller is returned a  cookie) to allow
318 1 *  continuing on the next call to this function.
319 1 *
320 1 *  Parameters:
321 1 *     context
322 1 *     parentObject   (I)  -  Pointer to the restore context
323 1 *     objectLevel    (I)  -  the parent object
      1 *                     (I)  -  specifies whether the parent object is a top level or
      1 *                             container object
```

```
 *    objects
 *              (  O)  - a pointer to receive the start of the objects list
 *    cookie    (IO)  - a place holder for the list position
 *    maxEntries (I)  - the maximum number of objects to return
 *    numberBadFiles (O) - the real number of objects returned in the list
 *
 *    allowBadFiles (I) - flag whether or not to include bad files
 *
 *    Returns:
 *    E_SUCCESS          on success
 *    EP_RB_RECOVER_xxx  on error
 *
 *    errno_by
 *    plug-in            on success
 *                       on error
 *
 ***********************************************************/
RSTPl_GetNextLevelObjects( restore_context,
                           restoreableObjects
                           enum RSTPl_Objects_Level,
                           struct RSTPl_uro_list
                           long               *objects,
                           const long         *cookie,
                           long               maxEntries,
                           const boolean_ty   allowBadFiles );

/***********************************************************
 *    Clear Restore Context :
 *
 *    This function is called to notify the plug-in that is currently
 *    top level object is no longer selected for browsing and marking. The
 *    plug-in should perform whatever cleaning and memory deallocation is
 *    appropriate.
 *
 *    Returns:
 *    E_SUCCESS          on success
 *    EP_RB_RECOVER_xxx  on error
 *
 *    errno_by
 *    plug-in            on success
 *                       on error
 *
 *    Parameters:
 *    context   (I)  - Pointer to the restore context
 *
 ***********************************************************/
RSTPl_ClearRestoreContext( restore_context *context );

/***********************************************************
 *    Submit
 *
 *    This function creates a submit object from the currently marked
 *    restoreable objects.
 *
 *    EDMRST_Start to begin execution of the restore.
 *
 *    Parameters:
 *    context   (I)  - Pointer to the restore context
 *    hostName  (I)  - host to restore to (only if inPlace == False)
 *    policy    (I)  - The overwrite policy to use
 *    inPlace   (I)  - Flag if the restore is to be in original locations
 *                       only if inPlace == False)
 *    directory (I)  - directory to restore to (
 *    transport (I)  - Indicator of transport the restore is to be over (SCSI
 *                       or network)
```

```
 *    submittObjlDptr (IO) - ID of the submit user object created to describe
 *                           the restore
 *    progressCB      (I)  - pointer to callback function to report progress and
 *                           test for cancellation
 *
 *    StartRestore
 *
 *    errno_by RSTPl_Submit( restore_context       *context,
 *                           const char            *hostName,
 *                           const char             policy,
 *                           const boolean_ty       inPlace,
 *                           const char             OverwritePolicy,
 *                           const char            *directory,
 *                           const char            *transport,
 *                           const RestoreTransport *submittObjlDptr,
 *                           int RSTPl_SubmitProgressProc progressCB );
 *
 *    This optional function begins the application-specific execution of
 *    restore of the objects in a submit object. The quiet/test callback
 *    function must be called periodically to check for cancellation of
 *    the restore.
 *
 ***********************************************************/
RSTPl_StartRestore( restore_context,
                    int              restore_context,
                    submitDptr,
                    quietSrcCB );

/***********************************************************
 *    Find Routines.
 *
 *    Parameters:
 *    context   (I)  - Pointer to the restore context
 *
 *    errno_by RSTPl_StartRestore( restore_context,
 *                                 submittObjlDptr,
 *                                 quietSrcCB ) ;
 *
 *    This function begins the
 *
 *    quietSrcCB  (I)  - function to call to check for the quit signal
 *
 ***********************************************************/

/***********************************************************
 *    These two functions allow the user to find restorable objects.
 *
 *    context   (I)  - Pointer to the restore context
 *
 *    is a linked list of found objects. Each entry in the list contains a
 *    pointer to a restorable object, and the backup time associated with the
 *    object.
 *
 *    The find operation is performed by the asynchronous thread of the
 *    engine, with the RSTPl_FindRestorableObjects function cooling
 *    RSTPl_FindRestorableObjects if a plug-in has its own find logical.
 *    RSTPl_GetFindResults is used to test for completion of the find.
 *
 *    signal cancellation. A callback function pointer is passed into
 *    RSTPl_FindRestorableObjects to allow testing for cancellation
 *    and to pass back progress information (
 *                          a count of the objects found so far.
 *    The second service library function, RSTPl_GetFindResults is also used to
```

```
433 1 *     retrieve the output of the find operation.
434 1 *
435 1 *     RSTPL_FindRestorableObjects Parameters:
436 1 *
437 1 *      context          (I) - Pointer to the restore context.
438 1 *      searchCriteria   (I) - The criteria used for the search.
439 1 *      intr_callback    (I) - Function to check for cancel and return progress
440 1 *
441 1 errno_t
442 1 RSTPL_FindRestorableObjects( restore_context    *context,
443 1                              RSRBC_SearchCriteriae  *searchCriteria,
444 1                              intr_callback          intr_callback );
445 1 *
446 1 *     RSTPL_GetFindResults Parameters:
447 1 *
448 1 *      context          (I) - Pointer to the restore context.
449 1 *      cancel           (I) - requests cancelation of the find (if TRUE)
450 1 *      maxEntries       (I) - the maximum number of objects to return
451 1 *      foundObjs        (O) - a pointer to a linked list of found objects
452 1 *      numberEntries    (O) - the real number of objects returned in the list.
453 1 *      cookie           (IO) - a place holder for excess found objects
454 1 *
455 1 errno_t
456 1 RSTPL_GetFindResults( restore_context    *context,
457 1                       const boolean_t    cancel,
458 1                       const long         maxEntries,
459 1                       struct RSTRV_found_obj_list  *foundObjects,
460 1                       long               *numberEntries,
461 1                       long               *cookie );
462 1
463 1
464 1
465 1
466 1
467 1 /*********************************************************************
468 1 * Get Top Level Templates.
469 1 *
470 1 * This function is required to retrieve the templates with which a
471 1 * object could have been backed up.
472 1 *
473 1 * Parameters:
474 1 *      context          (I) - the restore context
475 1 *      topLevelObj      (I) - Pointer to the top level object
476 1 *      templates        (O) - a pointer to the start of the list of templates
477 1 *      numberEntries    (O) - the real number of templates returned in the list
478 1 *
479 1 errno_t
480 1 RSTPL_GetTopLevelTemplates( restore_context    *context,
481 1                             struct RSTRPC_top_level_obj  *topLevelObj,
482 1                             struct RSTRV_name_list  **templates,
483 1                             short              *numberEntries );
484 1
485 1
486 1
487 1 /*********************************************************************
488 1 * Does Alternate Exist
489 1 *
490 1 *
```

```
491 1 *     This routine determines if an alternate trailset exists for the
492 1 *     given template.
493 1 *
494 1 *     Parameters:
495 1 *      context          (I) - Pointer to the restore context.
496 1 *      templateName     (I) - The name of the template to look for
497 1 *      exists           (O) - Return flag for whether or not the alternate exists
498 1 *
499 1 errno_t
500 1 RSTPL_DoesAlternateExist( restore_context    *context,
501 1                           const template_name_ty  templateName,
502 1                           boolean_t          *exists );
503 1
504 1
505 1 /*********************************************************************
506 1 * Mark Object
507 1 *
508 1 * The MarkObject operation takes a restorableObject and marks it, and
509 1 * possibly its descendant files for restoral based on the input
510 1 *      criteria
511 1 * Since the RSTPL_MarkObject call is an asynchronously executed
512 1 *      operation
513 1 * in the Restore Engine that performs the marking, this function must
514 1 * periodically check for user-signaled cancelation,  and update progress
515 1 * data using the progress callback function argument.
516 1 *
517 1 * NOTE: This function is responsible for keeping the volumes needed
518 1 *      (abvlist) element of the Restore context up to date.
519 1 *
520 1 * Parameters:
521 1 *      context          (I) - Pointer to the restore context
522 1 *      thisObject       (I) - The restoral object.  can be a leaf object (e.g. a
523 1 *                             file), or a container object (e.g. a directory).
524 1 *      allowBadfiles    (I) - Should mark operation descend on the content
525 1 *      descend                of container objects.
526 1 *      allowBadfiles    (I) - allows marking of files or files with BADDATA.
527 1 *      BadFileCount     (O) - return the file count with BADDATA.
528 1 *      permDenyFiles    (O) - return the file count with permission denied.
529 1 *      fileMarked       (O) - return the file marked after this mark occurred.
530 1 *      lenMarkedFiles   (O) - return the length of files marked after this mark
531 1 *                             occurred.
532 1 *      dirMarked        (O) - return the total directories marked after this mark
533 1 *                             occurred.
534 1 *      otherMarked      (O) - return the total "other" files marked after this
535 1 *                             mark occurred.
536 1 *      progressCB       (I) - pointer to callback function to report progress and
537 1 *                             test for cancelation
538 1 *
539 1 errno_t
540 1 RSTPL_MarkObject( restore_context    *context,
541 1
```

```c
                    struct RSTRPC_user_restorable_object *thisObject,
                    boolean_ty          allowadFiles,
                    boolean_ty          descend,
                    unsigned long       *BadFilesCount,
                    unsigned long       *PermanyFilesCount,
                    u_hyper             *ItemMarkedFiles,
                    unsigned long       filesMarked,
                    unsigned long       *dirMarked,
                    unsigned long       *otherMarked );
                    RSTPL_MarkProgressProc progress3 );
```

```
/*
 * UnmarkObject
 *
 *     The UnmarkObject operation takes a restorableObject and unmarks
 *     possibly its descendent files for restoral based on the input
 *     criteria.
 *
 *     Since the RSTPL_UnmarkObject call is an asynchronously executed
 *     operation, the Restore Engine that performs the unmarking function
 *     must periodically check for user-signaled cancellation and update
 *     progress data using the progress callback function argument.
 *
 * NOTE: This functions is responsible for keeping the volumes needed
 *       list
 *       (abvlist) element of the restore context up to date.
 *
 * UnmarkObject Parameters:
 *
 *   context:        (I) - Pointer to the restore context
 *   thisObject:     (I) - The restoral object to operate on (e.g. a
 *                         file), or a container object ( i.e. a directory).
 *
 *   BadFilesOnly    (I) -
 *   descend         (I) - Should unmark operation descend to operate on the
 *                         content of container objects.
 *   filesMarked           return the file count with BADDATA.
 *   ItemMarkedFiles
 *                   (I) - return the total files marked after this unmark
 *   dirMarked
 *                   (O) - return the length of files marked after this unmark
 *   otherMarked
 *                   (O) - return the total directories marked after this unmark
 *
 *                   (I) - return the total "other" files marked after this unmark
 *   progress3       (I) - pointer to callback function to report progress and
 *                         test for cancellation
 */
```

```
/*
 * IsObjectMarked
 *
 *   Is Object Marked
 *
 *     This function determines if a restorableObject has been
 *     marked for restoration.
 *     It is intended to allow the user to determine the
 *     current restore markings for the restorable objects at each
 *     hierarchy level, i.e. objects that have the same parent restorableObject.
 *
 * IsObjectMarked Parameters:
 *
 *   context:        (I) - Pointer to the restore context
 *   thisObject:     (I) - The restoral object to be checked; can be a leaf object
 *                         (e.g. a file), or a container object ( i.e. a directory).
 *
 *   marked          (O) - boolean to receive the marked / unmarked result
 */
 errno_ty RSTPL_IsObjectMarked( restore_context            *context,
                                struct RSTRPC_user_restorable_object *thisObject,
                                boolean_ty                 *marked );
```

```
/*
 * IsObjectMarkable
 *
 *   Is Object Markable
 *
 * Function Description:
 *     returns TRUE if the specified object is markable by the user,
 *     returns FALSE if it is not. This function applies only to
 *     container (directory) and leaf (file) objects.
 *
 * Parameters:
 *   context         (I) - Pointer to the restore context
 *   thisObject:     (I) - ptr to the restorableObject in question
 *
 * Return:
 *     TRUE  - the specified object is markable by the user
 *     FALSE - the specified object is not markable by the user
 */
 boolean_ty RSTPL_IsObjectMarkable( restore_context           *context,
                                    struct RSTRPC_user_restorable_object *thisObject );
```

```
/*
 * Get Necessary Media:
 *
 *     This OPTIONAL function is provided to allow retrieval of the
 *     media necessary to restore the currently marked objects.
 *
 *     If a plugin does
 *     not supply this function, then it must maintain the abvlist volume list
 *     attached to the restore context whenever mark and unmark are
 *     called, so that
 *     the generic RSTPL_GetNecessaryMedia function can retrieve the
 *     media list.
 */
```

```
647  *        Parameters:
648  *            context      (I) - Pointer to the restore context.
649  *            objects
650  *                     O) - address to return a pointer to the list of objects in
651  *            numberObjects
652  *                     O) - the real number of media objects returned in the array
653  *
654  ***********/
655  eerrno_ty RSTPL_GetNecessaryMedia( restore_context *context,
656                                     struct RSTPL_media_list **objects,
657                                     short           *numberEntries );
658
659  /***********
660  *
661  * RSTPL_GetAllBackupTimes
662  *
663  *    Function Description:
664  *        Get All Backup Times
665  *
666  *        Retrieve the dates of the backups within the time range
667  *        specified by the caller.
668  *
669  *    Parameters:
670  *        context      (I) - Pointer to the restore context
671  *        startTime        - include no earlier than this date
672  *        endTime          - include no later than this date
673  *        flags        (I) Backup constraint flags: e.g. full-only/partial-ok
674  *        timeList     (O) ptr to linked list of times
675  *        numEntries   (O) count of times returned
676  *
677  *    Return Codes:
678  *        E_SUCCESS        - operation completed successfully
679  *
680  ***********/
681  eerrno_ty RSTPL_GetAllBackupTimes( restore_context *context,
682                                     const time_t    startTime,
683                                     const time_t    endTime,
684                                     RSTPL_backup_flags_ty flags,
685                                     struct RSTPL_time_list  *timeList,
686                                     short           *numEntries );
687
688  /***********
689  *
690  * RSTPL_GetCurrentBackupTime
691  *
692  *    Function Description:
693  *        Retrieve the time of the backup that the current restore context
694  *        is set to and return it in the preallocated buffer.
695  *
696  *    Parameters:
697  *        context      (I) the time of the backup
698  *        bkupTime     - (I) Pointer to the restore context
699  *
700  *    Return Codes:
701  *        E_SUCCESS                   - operation completed successfully
702  *        EP_RB_RECOVER_INV_ARGS      - call issued out of sequence
703  *        EP_RB_RECOVER_BAD_ARGS      - invalid input argument
704  *        EP_RB_RECOVER_NO_CURR_BACKUP - no valid backup currently
705  *
706  ***********/
707  eerrno_ty RSTPL_GetCurrentBackupTime( restore_context *context,
708                                        const time_t    *bkupTime );
709
```

```
711  * RSTPL_GetCurrentBackupTime( restore_context *context,
712  *                             const time_t    *bkupTime );
713  *
714  *                    that is before the specified time.
715  *                    if an exact match is not possible.
716  *                    or the most recent
717  *
718  ***********/
719
720  /***********
721  *
722  * RSTPL_SetBackupForTime
723  *
724  *    Function Description:
725  *        Set the backup plane of the specified time.
726  *        switch to the backup plane of the specified time
727  *
728  ***********/
729  eerrno_ty RSTPL_SetBackupForTime( restore_context *context,
730                                    const time_t    forTime,
731                                    RSTPL_backup_flags_ty flags );
732
733  /***********
734  *
735  * RSTPL_SetCurrentBackup
736  *
737  *    Function Description:
738  *        Set the restore context to that of the current backup with
739  *        to the current one.                                   respect
740  *
741  *    Parameters:
742  *        context      (I) - Pointer to the restore context
743  *        forTime      (I) - The time for which the backup is requested
744  *        flags        (I) - Backup constraint flags: e.g. full-only/partial-ok
745  *
746  *    Return Codes:
747  *        E_SUCCESS                       - operation completed successfully
748  *        EP_RB_PREV_CATALOG              - when at the first catalog
749  *        EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the file
750  *
751  ***********/
752  eerrno_ty RSTPL_SetPrevBackup( restore_context *context,
753                                 RSTPL_backup_flags_ty flags );
754
755  /***********
756  *
757  * RSTPL_SetNextBackup
758  *
759  *    Function Description:
760  *        This routine must set the restore environment to the the next
```

```
the new catalog
 *                                                           *
 *                                                           *
761  *   Parameters:                                          *
762  *      context:   (I) - Pointer to the restore context   *
763  *      flags                                             *
764  *               )                                        *
765  *                                 *context,              *
766  *                                 flags );               *
767  *   Return Codes:                                        *
768  *      R_SUCCESS                                          *
769  *      EP_RB_RECOVER_NO_NEXT_CATALOG - when at the most recent *
                                           catalog
 *****************************************************/
771  eerrno_ty
772  RSTPL_SetNextBackup( restore_context       *context,
773                       RSTPC_backup_flags_ty  flags );

775  /*************************************************************
776  *                                                           *
777  *   Function Description:                                    *
779  *      Set the restore context to that of the first backup plane. *
780  *                                                           *
781  *   Parameters:                                             *
782  *      context:   (I) - Pointer to the restore context      *
783  *      flags      (I) - Backup constraint flags; e.g., full-only/partial-ok *
784  *                                                           *
785  *   Return Codes:                                           *
786  *      R_SUCCESS  - operation completed successfully        *
787  *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the *
                                             file
 *****************************************************/
789  eerrno_ty
790  RSTPL_SetFirstBackup( restore_context       *context,
791                        RSTPC_backup_flags_ty  flags );
792
793  /*************************************************************
794  *   Set First Backup                                        *
795  *                                                           *
796  *   Function Description:                                    *
797  *      Set the restore context to that of the most recent backup *
798                                                       plane.
799  *                                                           *
800  *   Parameters:                                             *
801  *      context:   (I) - Pointer to the restore context      *
802  *      flags      (I) - Backup constraint flags; e.g., full-only/partial-ok *
803  *                                                           *
804  *   Set Most Recent Backup                                  *
805  eerrno_ty
806  RSTPL_SetMostRecentBackup( restore_context       *context,
807                             RSTPC_backup_flags_ty  flags );
808
809  *   Return Codes:                                           *
810  *      R_SUCCESS                                            *
811  *      EP_RB_RECOVER_NO_CATALOG - when at the new catalog   *
812  *                                                           *
813  *   Return Codes:                                           *
814  *      R_SUCCESS                                            *
815  *      EP_RB_RECOVER_PERMISSION_DENIED - when user cannot access the *
```

```
file of
 *                                                           *
818  *                                                         *
819  *                                                         *
820  eerrno_ty
     RSTPL_SetMostRecentBackup( restore_context       *context,
                                RSTPC_backup_flags_ty  flags );
822  *                                                         *
823  *                                                         *
824  *                                                         *
826  *   Parameters:                                           *
827  *      context:   (I) - Pointer to the restore context    *
828  *      flags      (I) - Backup constraint flags; e.g., full-only/partial-ok *
829  *                                                         *
830  *                       of the new catalog                *
831  *   Function Description:                                  *
832  *      Determine if a backup exists prior to the backup that is *
833                              currently selected.
835  *                                                         *
836  *   Parameters:                                           *
837  *      context:   (I) - Pointer to the restore context    *
838  *      flags      (I) /                                   *
840  *   Return Codes:                                         *
841  *      R_SUCCESS                                          *
842  *      EP_RB_RECOVER_xxx                                  *
 *****************************************************/
844  *                                                         *
845  *   Is There Prev Backup                                  *
847  /*************************************************************
849  *                                                         *
850  *   Is There Next Backup                                  *
851  *                                                         *
852  *   Function Description:                                  *
853  *      Determine if a backup exists after the backup that is *
854                              currently selected.
856  *   Parameters:                                           *
857  *      context:   (I) - Pointer to the restore context    *
858  *      flags      (I) - Backup constraint flags; e.g., full-only/partial-ok *
859  *      isThere    (O) - TRUE/FALSE that requested backup does exist *
860  *                                                         *
861  *   Return Codes:                                         *
862  *      R_SUCCESS                                          *
863  *      EP_RB_RECOVER_xxx                                  *
864  *                                                         *
865  eerrno_ty
866  RSTPL_IsThereNextBackup( restore_context       *context,
                               RSTPC_backup_flags_ty  flags,
                               boolean_ty            *isThere );
867  *                                                         *
869  eerrno_ty
870  RSTPL_IsTherePrevBackup( restore_context       *context,
                               RSTPC_backup_flags_ty  flags,
                               boolean_ty            *isThere );
 *****************************************************/
872  *                                                         *
873  *   Is There Prev Backup For Time                         *
 *                                                           *
```

```c
874  1   /*
875  1    * Function Description:
876  1    *    Determine if a backup exists prior to the specified time
877  1    *
878  1    * Parameters:
879  1    *    context,       (I) - Pointer to the restore context
880  1    *    thisTime (I) - Time for the query
881  1    *    flags          (I) - Backup constraint flags; e.g., full-only/partial-ok
882  1    *    isThere (O)  - TRUE/FALSE that requested backup does exist
883  1    *
884  1    * Return Codes:
885  1    *    E_SUCCESS      - operation completed successfully
886  1    *    EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
887  1    *
888  1    ******************************************************/
889  1   errno_ty
890  1   RSTPL_IsTherePrevBackupForTime( restore_context        *context,
891  1                                   const time_t            thisTime,
892  1                                   RSTRPC_backup_flags_ty  flags,
893  1                                   boolean_ty             *isThere );
894  1
895  1
896  1   /*
897  1    * Is There NextBackup For Time
898  1    *
899  1    * Function Description:
890  1    *    Determine if a backup exists after to the specified time
901  1    *
902  1    * Parameters:
903  1    *    context      (I) - Pointer to the restore context
904  1    *    thisTime (I) - time for the query
905  1    *    flags        (I) - Backup constraint flags; e.g., full-only/partial-ok
906  1    *    isThere (O)  - TRUE/FALSE that requested backup does exist
907  1    *
908  1    * Return Codes:
909  1    *    E_SUCCESS      - operation completed successfully
910  1    *    EP_RB_RECOVER_xxx  - when errors occur accessing catalogs
911  1    *
912  1    */
913  1   errno_ty
914  1   RSTPL_IsThereNextBackupForTime( restore_context        *context,
915  1                                   const time_t            thisTime,
916  1                                   RSTRPC_backup_flags_ty  flags,
917  1                                   boolean_ty             *isThere );
918  1
920  1   #ifdef __cplusplus
921  1   }
922  1   #endif
924      #endif   /* #ifndef H_RSLPIAPI */
```